VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Webová aplikace pro podporu analýzy biomedicínských dat

Web Application for Support of Biomedical Data Analysis

Bc. Radovan Pranda

Diplomová práce

Vedoucí práce: doc. Ing. Petr Gajdoš, Ph.D.

Ostrava, 2021

Abstrakt

Cílem této práce je vytvořit webovou aplikaci, která má sloužit ke spouštění skriptů pro analýzu biomedicínských dat na výpočetním serveru. První část práce je proto věnována průzkumu technologií, které by mohly být použity při vývoji aplikace, především zmapování nástrojů pro spouštění, správu procesů a přidělování zdrojů procesům v prostředí HPC. Speciální pozornost je věnována Son of Grid Engine, principům jeho fungování, omezením a rozhraním umožňujícím komunikaci s ním. Kromě toho je také věnována knihovnám, které umožňují generovat formuláře z objektu typu JSON, a průzkumu jimi poskytovaných funkci.

Druhá část práce se zabývá návrhem a implementací výsledné aplikace. V rámci ní jsou rozebírány i návrh a implementace knihovny, umožňující generování formulářů z objektu JSON, která je vytvořena přímo na míru dle požadavků výsledné webové aplikace. Poslední část práce se věnuje testování výsledné aplikace.

Kľúčové slová

Grid engine, Sun grid engine, Son of Grid engine, ASP .NET Core Web API, React, generování formulářů, plánovaní jobů, management zdrojů

Abstract

The object of this diploma thesis is to create a web application that should be used to run scripts for the analysis of biomedical data on a computing server. The first part of the thesis is therefore devoted to the research of technologies that could be used in the development of the resulting application, especially the mapping of tools for process and resource management in HPC environment (job schedulers). Special attention is paid to the Son of Grid Engine, the principles of its operation, limitations and interfaces enabling communication with it. In addition, it is also dedicated to libraries that allow you to generate forms from a JSON object and explore the functions they provide.

The second part deals with the design and implementation of the resulting application. It also discusses the design and implementation of a library that allows the generation of forms from the JSON object, which was created directly according to the requirements of the resulting web application. The last part of the thesis is devoted to testing the final application.

Keywords

Grid engine, Sun grid engine, Son of Grid engine, ASP .NET Core Web API, React, form generator, job scheduling, resource management

Poďakovanie

Rád by som poďakoval doc. Ing. Petru Gajdošovi, Ph.D. za odbornú pomoc a konzultáciu pri vytváraní tejto diplomovej práce, a Ing. Danielovi Stříbnemu za poskytnutie zariadenia na testovanie a prístupu k HPC.

Obsah

Zoznam použitých skratiek a symbolov

API	–	Application Programming Interface
AWS	–	Amazon Web Services
BLOB	–	Binary Large Object
CPU	–	Central processing unit
DDOS	–	Distributed Denial-of-Service
DRM	–	Distributed Resource Management
DRMAA	–	Distributed Resource Management Application API
DRMS	–	Distributed Resource Management System
FIFO	–	First in first out
HPC	–	High Performance Computing
HTTP	–	Hypertext Transfer Protocol
JS	–	JavaScript
JSON	–	JavaScript Object Notation
MPI	–	Message Passing Interface
MVC	–	Model-View-Controller
NPM	–	Node Package Manager
OGE	–	Oracle grid engine
OGS	–	Open Grid Scheduler
OS	–	Operation system
PC	–	Personal computer
PBI	–	Python Buffer Interface
QMON	–	Queue Monitor
RESTful	–	Representational State Transfer
SGE	–	Sun grid engine
SHA	–	Secure Hash Algorithm
SoGE	–	Son of Grid engine
SSH	–	Secure Shell
SWIG	–	Simplified Wrapper and Interface Generator

TCP	–	Transmission Control Protocol
UGE	–	UNIVA grid engine
UI	–	User Interface
VM	–	Virtual Machine
VPN	–	Virtual Private Network
WLA	–	Workload Automation

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

Od vzniku prvého počítača patrili k hlavným problémom nedostatok a obmedzenosť zdrojov, ako aj efektivita ich využitia pri riešení rôznych úloh. Postupom času sa síce mnohonásobne zvýšila rýchlosť hardwaru, omnoho viac a rýchlejšie však vzrástlo množstvo problémov, ktoré sa ľudia snažili vyriešiť a veľkosť dát, ktoré je potrebné pri ich riešení spracovať. Tým stúpli aj nároky na efektivitu plánovania a vykonávania jobov, obzvlášť u HPC, ktoré spracúvavajú najväčšie množstvo dát. Tá je závislá z veľkej časti na plánovačoch, medzi ktoré patrí aj Grid engine, ktorého možnosti a limity budú preskúmané v úvode prvej kapitoly venovanej použitým technológiam.

Medzi odvetvia s najväčším množstvom dát potrebným na spracovanie patrí tiež medicína. Preto je cieľom tejto práce vytvoriť webovú aplikáciu, ktorá by umožňovala čo najefektívnejšie plánovanie pipeline tvorených skriptami slúžiacimi na analýzu biomedicínskych dát spúšťaných vo forme jobov prostedníctvom Grid enginu. Keďže každý zo skriptov môže mať iné požiadavky na software alebo hardware potrebné pre svoj beh, užívateľ by mal byť schopný ich zadefinovať, aby bolo využívanie zdrojov, ktorými HPC disponuje, čo najefektívnejšie. Okrem toho môže pôžadovať každý zo skriptov iné vstupné parametre líšiace sa hodnotami, dátovým typom alebo počtom. Je preto potrebné nájsť jednoduchý a efektívny spôsob, akým generovať formuláre, ktoré by umožnili parametre zadávať a v prípade pridávania skriptu do systému tiež umožniť takýto formulár nadesignovať.

Pretože je rozsiahlosť riešených problémov veľká, častokrát ich riešia skupiny jobov, ktoré na seba môžu naväzovať a odovzdávať si medzi sebou dáta tvoriac tak pipeline. Všetkým z týchto spomínaných problémov sa bude venovať druhá kapitola. Osobitná pozornosť bude venovavá tiež bezpečnosti, aby bol zabezpečený čo najmenší dopad skriptov na vykonávanie pipeline ostatných užívateľov. V neposlednom rade musí byť užívateľovi umožnené sledovať dianie na výpočtovom zariadení, dostupnosť zdrojov a aktuálne vyťaženie ako vrámci jednotlivých hostov, tak vrámci front, ktoré sú na nich umiestnené, aby sa mohol rozhodnúť, či chce na danom zariadení s Grid enginom pipeline naplánovať.

Vrámci poslednej kapitoly budú popísané výsledky testovania navrhovaného riešenia, požiadavky, ako aj problémy, ktoré nastali počas vývoja.

Kapitola 2

Použité technológie

Nasledujúca kapitola je venovaná prieskumu technológii použitých pri vývoji výslednej aplikácie. V úvodných podkapitolách budú rozoberané hlavne job management pomocou Grid enginu, princípy jeho fungovania a možnosti komunikácie prostredníctvom rôznych rozhraní, pričom špeciálna pozornosť bude venovaná knižniciam.

Druhá časť kapitoly predstavuje prieskum a porovnanie balíčkov Node.js umožňujúcich generovanie formulárov z objektov JSON. Okrem porovnania vlastností, rozdielov, výhod a nevýhod balíčkov budú uvedené aj príklady ich použitia.

2.1 Grid engine

Plánovače a ich efektivita majú obrovský význam pri vykonávaní hardwarovo náročných úloh a spracovávaní veľkého množstva dát. V oblasti cluster computingu patrí medzi jeden z najvýznamnejších práve Grid engine. Jedná sa o počítačovú aplikáciu slúžiacu na riadenie vykonávania jobov, označovanú často aj ako batch scheduler, DRM, DRMS alebo WLA. V prípade Grid enginu sa konkrétne jedná o batch-queuing systém, teda systém radiaci joby do fronty. Plánovanie a následné vykonávanie teda prebieha systémom FIFO. Okrem Grid enginu existujú aj ďalšie plánovače, akými sú napríklad: OpenLava, schedulix, VisualCron, ... [1]

V súčasnej dobe existuje niekoľko komerčných riešení využívajúcich niektorú z verzii Grid enginu. Medzi najznámejšie komerčné riešenia v tomto smere patria:

- **Azure CycleCloud** - komplexné plnohodnotné riešenie využívajúce Open Grid Scheduler v kombinácii s ďalšími softwarovými riešeniami a kombinuje tak cloud computing s flexibilným prostredím HPC.[2]
- **Amazon Web Services** - jednou z poskytovaných služieb je aj možnosť cluster computingu využitím Univa Grid Engine, ktorý možno ľahko nainštalovať v AWS Marketplace. Nejedná sa však o tak komplexné riešenie ako v prípade Azure CycleCloud.[3]

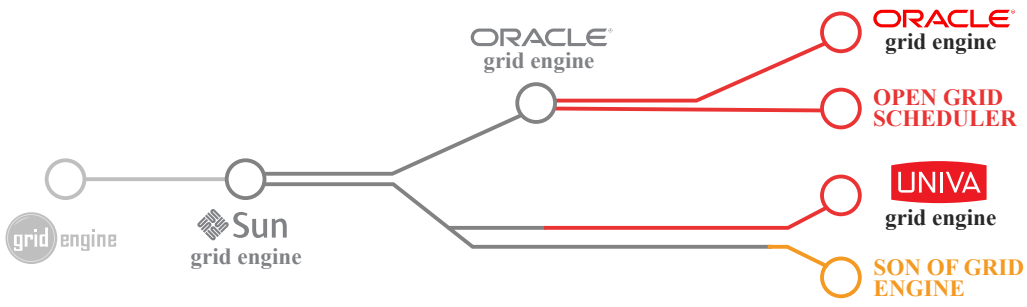
Okrem vyššie spomínaných riešení poskytuje svoje služby v oblasti cluster computingu aj Google vrámci Google Cloud s využitím ElastiCluster, ktorý je dostupný tiež ako služba od Amazonu či vrámci privátneho OpenStack cloudu.[4]

2.1.1 História

Ako vidieť na obrázku 2.1, všetky rozoberané grid enginy vznikli rozšírením a opravami softwaru Grid Engine. Ten v roku 2000 kúpila spoločnosť Sun Microsystems, ktorá zmenila názov enginu na Sun grid engine (ďalej len SGE). Vďaka popularite spoločnosti, dobrému dizajnu, schopnostiam a vlastnostiam systému sa rapídne zvýšila jeho popularita. Ešte pred samotným odkúpením spoločnosti Sun Microsystems spoločnosťou Oracle uvoľnila spoločnosť veľkú časť svojho portfólia zadarmo, pričom do tohto portfólia patril aj SGE.

Po odkúpení bol engine premenovaný na Oracle grid engine (ďalej len OGE). Oracle nemalo záujem o prácu na systéme pre HPC aj napriek tomu, že ho mohli začleniť do svojej verzie Linuxu a urobiť tak z neho hlavnú platformu pre HPC. Namiesto toho spoločnosť zakrátko uzatvorila komunitu, zastavila šírenie kódu SGE, znehodnotila stránky a zmazala publikácie v snahe zarobiť na engine čo najviac. Software začali predávať pod názvom Oracle Enterprise Manager 11G, ktorý plnil funkciu monitorovacieho systému pre Oracle databáze. Popri OGE, platenej verzii Grid enginu, vznikla tiež open source verzia založená na poslednej release verzii od Oracle, ktorá je známa pod názvom Open Grid Scheduler, skrátene OGS. [5]

V roku 2011 najala spoločnosť UNIVA vývojárov z tímu, ktorý tvoril SGE, aby vytvoril vlastnú odnož Grid enginu. Bola založená na SGE a nazvaná Univa grid engine (ďalej len UGE), pričom zahŕňala aj komerčnú podporu a stala sa veľmi významným konkurentom OGE. Známou sa však stala v komunite hlavne pre veľké množstvo bugov a opravných balíčkov, v čom dokonca prekonal OGS, ktorý sa síce potýkal s obdobným problémom, avšak v omnoho menšej miere. Po odkúpení práv od Oracle si spoločnosť UNIVA pokazila dobré meno okrem iného aj kvôli poplatkom, ktoré boli účtované formou 1 licencia na jadro procesora na rok.[6]



Obr. 2.1: Vývojová línia Grid enginu

Po neúspechu sa vyskytli rozpory v spoločnosti UNIVA. Tá mala v pláne predstaviť UGE open source core verziu, tento plán bol však zrušený, čo malo za následok odchod vývojára Dave Love zo

spoločnosti. Ten stojí za posledným známym nástupcom SGE, ktorým je Son of grid engine (ďalej len SoGE). Jedná sa o komunitný projekt, ktorý sa vyvinul z pôvodného SGE a je stále svojou komunitou udržiavaný a opravovaný. [5][7]

2.1.2 Daemoni

Samotný Grid engine má priamo integrované služby bežiace na pozadí, ktoré navzájom spolupracujú a starajú sa o základné funkcie, ale aj o to, aby bolo spúšťanie a plánovanie jobov čo najbezpečnejšie a najefektívnejšie. Počet daemonov sa líši v závislosti na type a verzii Grid enginu. V prípade SoGE sa vyskytujú 2 hlavní daemoni, ktorí sú vrámci niektorej literatúry označovaní aj ako agenti:

- **master daemon** - riadi celkové správanie Grid enginu v clusteri (spravuje hostov, fronty, joby, vyťaženie systému a užívateľske oprávnenia, vykonáva plánovanie jobov)

- **execution daemon** - riadi lokálne fronty zariadenia, na ktorom je spustený, vykonáva a riadi joby obdržané od master daemona, ktoré sa majú spustiť v týchto frontách

Okrem nich sa v prípade prvých verzii SGE uvádzajú ešte ďalší 2 daemoni, ktorí sa v prípade SoGE nespomínajú, nakoľko boli začlenení do vyššie spomínaných služieb, pretože pracujú na ich pozadí a užívateľ s nimi teda bežne priamo neprichádza do kontaktu. Ich hlavnou úlohou je zabezpečovať spoluprácu komponentov a komunikáciu medzi master a execution daemonom. Sú to:[8]

- **scheduler daemon** - udržuje aktuálne informácie o stave clustera a v spolupráci s master daemonom rozhoduje o tom, do ktorej fronty bude job zaradený. Samotnú akciu potom vykoná master daemon.

- **communication daemon** - stará sa o komunikáciu medzi daemonmi prostredníctvom TCP portu

- **shadow master daemon** - slúži na detekciu chýb u master daemona

Okrem vyššie spomínaných funkcii tvoria daemoni bezpečnostnú vrstvu Grid enginu. Jednou z výhod master a execution daemonov je, že po správnom nakonfigurovaní nemusia bežať na rovnakom zariadení, ale môžu bežať oddelene, teda každý na inom zariadení, ktoré sú navzájom sieťovo prepojené.

Najmenším článkom z pohľadu správy jobu predstavuje Shepherd agent, ktorý umožňuje nadradenému procesu kontrolovať jeden job prostredníctvom signálov, a tak realizovať checkpointing mechanizmus, rovnako ako aj napr. pozastavenie, povolenie, ukončenie jobu.[9]

2.1.3 Plánovanie a limity Grid enginu

Plánovanie jobov prebieha u Grid enginu formou radenia do front, pričom miera s akou možno pracovať s frontami je závislá na role, ktorá je uživateľovi pridelená. Vrámci enginu rozlišujeme role:

- **Manažér / Správca** - je rola s najväčšími právami a plnou kontrolou nad systémom. Podľa predvolených nastavení majú takéto oprávnenia superuseri všetkých administračných hostov.
- **Operátor** - má rovnaké práva ako manažér s výnimkou zmeny konfigurácie. Nemôže teda napr. pridávať, mazať či modifikovať fronty.
- **Vlastník** - vlastník fronty môže oproti užívateľovi navyše pozastaviť alebo povoliť frontu, ktorú vlastní, rovnako ako joby, ktoré v nej sú.
- **Užívateľ** - nemá žiadne práva spravovať cluster ani fronty. Môže však spravovať svoje joby - spúšťať, modifikovať, zrušiť, sledovať, atď.

Pokiaľ má teda užívateľ pridelené práva manažéra, môže modifikovať a vytvárať fronty kedykoľvek počas behu Grid enginu. Následne do nich môže ktokoľvek s právami užívateľa (alebo vyššími) zaradiť job do fronty naplánovaním v prípade, že je mu k danej fronte povolený prístup. Joby sa delia na základe stavu, v ktorom sa nachádzajú na:

- **čakajúce (pending)** - job bol naplánovaný a čaká na spustenie
- **bežiace (running)** - job sa momentálne vykonáva
- **odložené (suspended)**
- **zombie** - job, ktorého vykonávanie sa ukončilo, ale jeho zdroje ešte neboli uvoľnené

Pri naplánovaní sa job zaradí na koniec fronty, kde čaká v stave pending na priradenie execution hosta a fronty, ktorá je na ňom umiestnená. Testovanie preukázalo, že medzi naplánovaním a spustením jobu je časový rozdiel, ktorého dĺžku možno nastaviť vrámci konfigurácie fronty. Celkový maximálny počet paralelne bežiacich jobov je pritom vrámci systému obmedzený na počet, ktorý pri inštalácii vymedzí užívateľ. U SoGE je v závislosti na verzii prednastavené maximum 100 alebo 200 jobov. Podobne obmedzený je aj maximálny počet jobov vrámci jednotlivých front, ktorý je možno nastaviť počas vytvárania fronty alebo neskôr v konfigurácii definovaním počtu slotov, pokiaľ má užívateľ pridelené práva manažéra.

2.2 Správa jobov a zdrojov pomocou Grid enginu

Grid engine ponúka viaceré možnosti, ako môžno plánovať a spravovať joby, ktoré sú mu zaslané na spracovanie. Medzi ne patria príkazy volané z príkazového riadku či rozhranie v podobe desktopovej aplikácie QMON. Ďalšiu možnosť plánovania a manažmentu jobov poskytujú knižnice.

2.2.1 Manažment jobov z príkazového riadku

Jednou z možností správy jobov, kontrolovania chodu a vyťaženia zariadenia Grid enginom predstavuje rozhranie príkazového riadku. V tomto smere poskytuje Grid engine možnosť spravovať a plánovať joby, prideľovať, sledovať dostupné zdroje, spravovať užívateľov a podobne. Medzi najdôležitejšie príkazy patria:

- **qacct** - sumár informácii týkajúcich sa dokončených jobov a užívateľských účtov vrámci Grid enginu, akými sú napríklad systémový čas, názov hosta a fronty, atď.
- **qalter** - úprava naplánovaneho alebo už bežiaceho jobu
- **qconf** - umožňuje konfigurovať Grid engine a poskytuje informácie týkajúce sa konfigurácie
- **qdel** - možnosť zastaviť job
- **qhold** - pozastavenie jedného alebo skupiny jobov
- **qhost** - poskytuje podrobné informácie týkajúce sa execution hostov
- **qstat** - výstupom tohto príkazu sú komplexné informácie o joboch, ktoré čakajú na spustenie, ako aj o tých, ktorých vykonávanie už začalo
- **qsub** - naplánovanie jobu a pridelenie potrebných zdrojov (napr. počet CPU, pamäte a pod.)

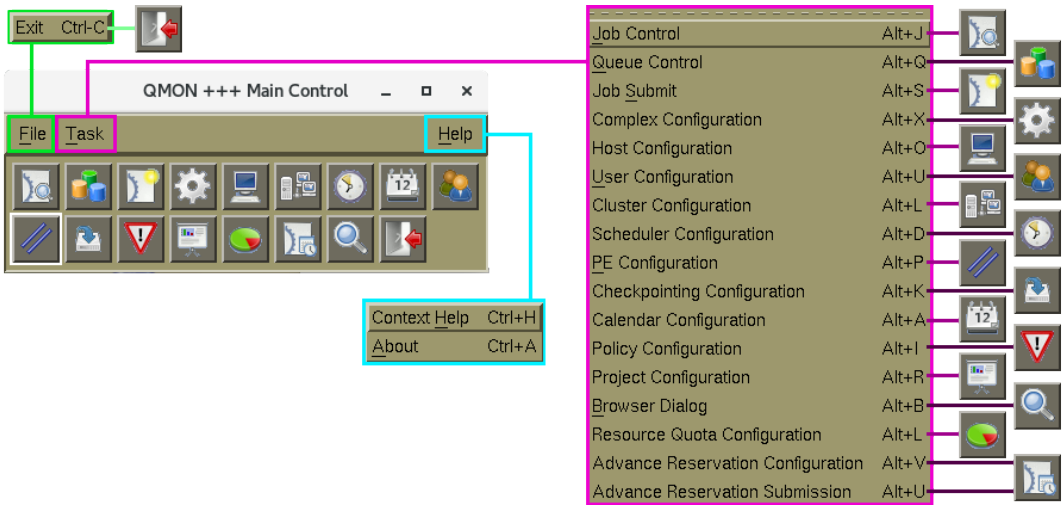
Okrem vyššie spomenutých príkazov existujú ešte príkazy: qselect, qrexec a qsh. Tieto príkazy obsahuje každá z vetiev Grid enginu, pričom každý z nich požaduje inú uživateľskú rolu.[8][10][11] Výhodou je, že niektoré z príkazov umožňujú serializovať výstup do XML objektu. Predvolene však vracajú výstup vo forme tabuľky.

2.2.2 Grafické užívateľské rozhranie

Ďalšiu z možností ovládania Grid enginu, plánovania a správy jobov predstavuje rozhranie desktopovej aplikácie, ktorú možno spustiť príkazom qmon. Toto rozhranie však nie je dostupné vrámci všetkých verzii Grid enginu a nie je súčasťou ani minimálnej inštalácie aplikácie. Aj to je dôvodom, prečo sa príkaz qmon, neuvádza v podkapitole 2.2.1.

Pomocou tejto aplikácie môžno naplánovať nový job, upravovať a rušiť joby vo fronte, ako aj spravovať samotné fronty a prideľovať im priority bez hlbšej znalosti Grid enginu a jeho príkazov. Umožňuje spravovať užívateľov, ktorí majú k enginu prístup, ich role a tým aj ich práva. Ďalšou výhodou je jednoduché, prehľadné zobrazenie čakajúcich, spracovávaných i dokončených jobov, informácii týkajúcich sa pridelených zdrojov a ďalších užitočných informácii.

Okrem vyššie spomenutých plusov má aplikácia i svoje mínusy. Medzi ne patrí napríklad chýbajúci log, ktorý je realizovaný formou textového súboru a pre neznalého užívateľa je pomerne ťažko



Obr. 2.2: Ukážka aplikácie QMON

dohľadateľný, ale aj nižšia výraznosť v prípade zobrazovania chybových hlášok spôsobená nevhodnou voľbou farby pozadia aplikácie, ktorá sa však verziu od verzie líši. Prehľadnejšou verziou tohto prostredia disponuje aj UGE a OGE.

2.2.3 Manažment jobov pomocou knižníc

Poslednú známu možnosť správy Grid enginu a komunikácie s ním predstavujú knižnice. Najvýznamnejšou z nich je DRMAA označovaná aj ako libdrmaa a to hlavne z dôvodu, že Grid engine obsahuje moduly umožňujúce spoluprácu práve s touto knižnicou. Okrem nej je veľmi známa a všeobecne významná aj knižnica MPI.

Z prieskumu, ktorého výsledky môžno vidieť v tabuľke 2.1 tiež vyplýva, že najväčšie množstvo knižníc je dostupných v jazyku Python 2, ktorého podpora skončila na začiatku roku 2020. Existuje však viacero spôsobov, ako možno tieto knižnice sprístupniť pre novšie verzie jazyka. Konkrétne sa jedná o portovanie kódu[12] a vzhľadom na veľmi malé rozdiely medzi verziami je možný aj automatizovaný preklad kódu z verzie 2.x na verziu 3.x.[13] Jednou z hlavných nevýhod týchto knižníc je to, že jazyk Python patrí medzi interpretované jazyky, ktorých vykonávanie kódu je pomalšie ako v prípade jazykov kompilovaných. Ďalšou z nevýhod týchto knižníc je, že sa v prevažnej väčšine jedná o wrappery iných knižníc.

Knižnice je možné rozdeliť do štyroch kategórii podľa použitého spôsobu komunikácie na:

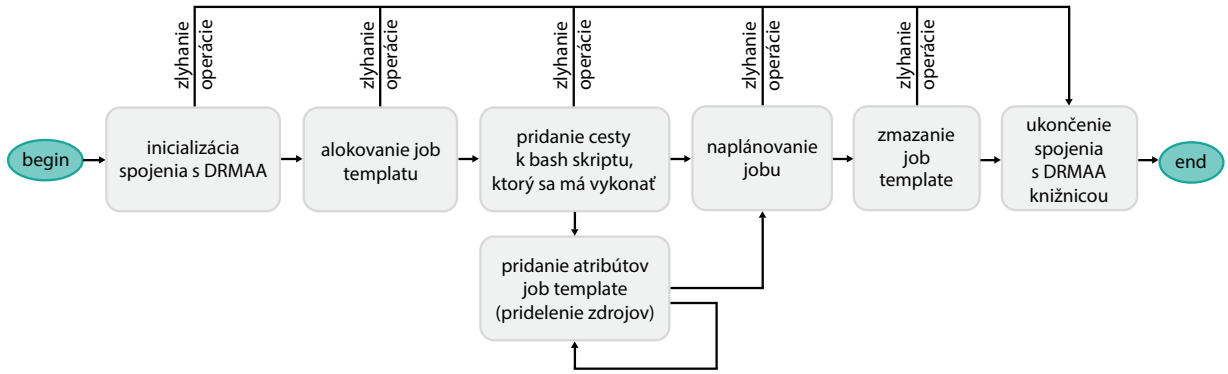
- knižnice využívajúce DRMAA
- knižnice využívajúce odovzdávanie správ (message passing)
- knižnice pracujúce s Grid enginom prostredníctvom rozhrania príkazového riadku
- wrappery - knižnice zaobaľujúce inú knižnicu

	Jazyk						
Názov	Python 2	Python 3	Java	C++	C	Go	Typ
Drmaa	•						wrapper
hbristow/gridengine	•						wrapper
Drmaa-2	•						API
QPy	•						využíva príkazový riadok
Mpi	•	•		•	•		message passing
Pygridtools/gridmap	•						wrapper
python_qsub wrapper	•						wrapper
Go-drmaa						•	wrapper
DRMAA	•	•	•	•	•		API

Tabuľka 2.1: Porovnanie knižníc podľa jazyka a typu

DRMAA

Táto knižnica predstavuje aplikačné rozhranie umožňujúce zadávanie a riadenie jobov DRM, akým je napr. výpočtová infraštruktúra typu cluster alebo grid. Rozsah API pokrýva všetky high level funkcie, ktoré sú potrebné na odosielanie, riadenie a monitorovanie jobov, spotrebovaných a dostupných zdrojov v DRM systéme, ktorým je aj Grid engine.[14]



Obr. 2.3: Postup zaradenia jobov do fronty na vykonanie pomocou knižnice DRMAA

Ako vidieť na obrázku 2.3, každej práci s DRMAA predchádza inicializácia, pričom na záver je potrebné vyžiadať ukončenie spojenia. Každý z príkazov vracia počet chýb, ktoré nastanú pri jeho vykonávaní. V prípade, že je vrátená hodnota rovná DRMAA_ERRNO_SUCCESS, príkaz sa vykonal úspešne. V opačnom prípade nastala chyba a algoritmus by nemal pokračovať ďalším krokom. Pre jednoduchosť a ľahšie pochopenie boli kontroly chýb zo schémy vypustené.

Jednou z nevýhod tejto knižnice je, že neumožňuje konfiguráciu samotného enginu, ale iba správu jobov vo frontách, čo je pochopiteľné vzhľadom k tomu, že knižnica nie je priamo určená pre Grid

engine, ale obecne na správu zdrojov zariadenia. Konkrétne možno teda prostredníctvom DRMAA joby relatívne jednoducho spúšťať, sledovať ich priebeh i zastaviť. Pre možnosť konfigurácie by bolo potrebné knižnicu rozšíriť. Samotná konfigurácia alebo rekonfigurácia za behu by mohla prebiehať iba priamou úpravou konfiguračných súborov enginu, pričom pre aplikovanie zmien je potrebné zavolať príkaz qconf v prostredí terminálu alebo reštartovať oboch daemonov.

Knižnica DRMAA je dostupná v jazyku C, ale vo forme bindingov aj v jazykoch: Java, JavaScript, C++, Go, Perl, Python, Ruby a Tcl. Pre ostatné jazyky je dostupné riešenie vo forme knižníc generovaných prostredníctvom SWIG.

Wrappery

Ako vidieť v tabuľke 2.1, najpočetnejšiu skupinu preskúmaných knižníc tvorili práve wrappery. Hlavnou nevýhodou wrapperov je, že ich fungovanie plne závisí na prítomnosti knižnice, ktorú zaobaľujú. Mnohé z nich pritom iba kopírujú štruktúru zaobalenej knižnice a nijak knižnicu nerozširujú. Spomedzi nájdených knižníc medzi ne patrí napríklad knižnica Go-drmaa.

V mnohých prípadoch wrappery funkcionalitu zaobaľovanej knižnice dokonca redukovali, čo predstavuje výhodu z hľadiska veľkosti wrappera, avšak nevýhodu z hľadiska poskytovanej funkčnosti. Hlavnou prednosťou wrapperov je totiž možnosť rozšírenia o funkcionality, ktoré nie sú dostupné v zaobaľovanej knižnici. Najväčší počet preskúmaných wrapperov zaobaľoval práve knižnicu DRMAA. Väčšinou pritom obsahovali implementáciu iba základných príkazov slúžiacich na interakciu s Grid enginom. Vo väčšine týchto prípadov sa jednalo len o základné príkazy: qsub, qhost, qstat a qdel, pričom v niektorých prípadoch boli nimi poskytované funkcie a možnosti výrazne okresané. Medzi takéto knižnice patria napríklad knižnice: python_qsub_wrapper a pygridtools/gridmap.

Hlavnou výhodou wrapperov je, že obsahujú funkcie, ktoré by si programátor musel za normálnych okolností naprogramovať sám. Táto výhoda so sebou prináša aj radu nevýhod, akými sú napríklad:

- **nevyužité funkcie** - funkcie, ktoré sa nevyužijú, zbytočne pridávajú knižnici na veľkosti
- **chýbajúce funkcie** - absencia a okresanosť funkcii núti k tomu, aby si niektoré z nich doprogramoval programátor sám
- **vysoká miera závislosti na iných knižniciach** - napr. na libdrmaa, drmaa2, atď.
- **úplná závislosť na zaobaľovanej knižnici**

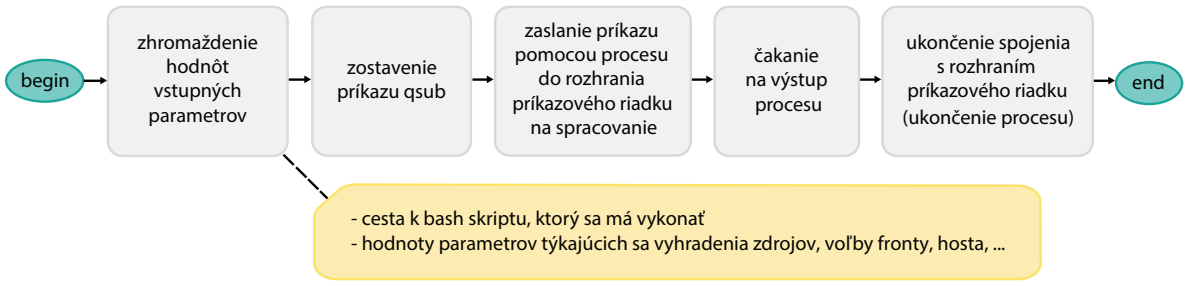
Vzhľadom na to, že sú moduly umožňujúce spoluprácu s API pre distribuovanú správu zdrojov súčasťou Grid enginu, nevýhoda úplnej závislosti na zaobaľovanej knižnici DRMAA nie je tak veľká ako v prípade zaobalenia inej knižnice. Presný opak predstavuje pár wrapperov, ktoré nezaobaľovali originálnu knižnicu, ale jej wrapper, ktorý ju rozširoval. Vo väčšine prípadov sa pritom jednalo

Názov knižnice	Obaľovaná knižnica	Druh	Detail
Drmaa	DRMAA	Výhody	poskytuje všetky funkcie knižnice DRMAA
		Nevýhody	nepridáva žiadne funkcie (iba kopíruje štruktúru DRMAA)
hbristow/gridengine	DRMAA	Výhody	odľahčená knižnica pre prácu s SGE
			obsahuje plánovače ProcessScheduler (pre debugovanie na viacjadrovom PC) a GridEngineScheduler (komunikácia s SGE)
		Nevýhody	kvôli dvom prístupom zaberá viac miesta
Pygridtools/gridmap	DRMAA	Výhody	kompatibilita s Python 3.x
			relatívne dobrá dokumentácia
		Nevýhody	závislosť na mnohých knižniciach
python_qsub wrapper	DRMAA	Výhody	možnosť použiť R modul
		Nevýhody	primárne dizajnovaný pre Sheffield cluster Iceberg
			používa len 3 funkcie
			podpora R 3.2.1 (testy pritom ukázali, že by mohol bežať aj s verziou 3.4.4)
			podpora len pre Python 2.x
Go-drmaa	DRMAA2	Výhody	v prípade rozšírenia by mohla podporovať naplánovanie jobu, monitorovanie clustera a job workflow management
		Nevýhody	nepridáva žiadne funkcie (iba kopíruje štruktúru DRMAA)

Tabuľka 2.2: Popis výhod a nevýhod preskúmaných wrapperov

o rozšírenie wrappera drmaa2. Okrem spomínaných všeobecných výhod a nevýhod môžno vidieť ďalšie popísané plusy a mínusy jednotlivých wrapperov vyššie v tabuľke 2.2.

Knižnice zasielajúce príkazy do prostredia príkazového riadku



Obr. 2.4: Schéma fungovania knižnice využívajúcej príkazový riadok

Počas prieskumu bola nájdená aj knižnica komunikujúca s enginom zasielaním príkazov do prostredia príkazového riadku, viď obrázok 2.4. Konkrétne sa jedná o knižnicu QPy v jazyku Python, ktorá bola robená namieru pre cluster Shanghai Jiao Tong University. [15] Na základe analýzy kódu sa podarilo zistiť, že táto knižnica umožňuje iba plánovanie jobov využitím príkazu qsub. Knižnica neposkytuje funkcie pre úpravu či zastavovanie jobov a neobsahuje ani možnosť Grid engine konfigurovať, čo knižnici značne uberá na využití a význame. Okrem veľmi okresanej funkčnosti patrí medzi nevýhody tejto knižnice aj to, že neposkytuje možnosť komunikácie prostredníctvom SSH.

Obecne pritom platí, že hlavnou výhodou takto komunikujúcich knižníc je, že na rozdiel od ostatných knižníc majú minimálnu závislosť na iných knižniciach.

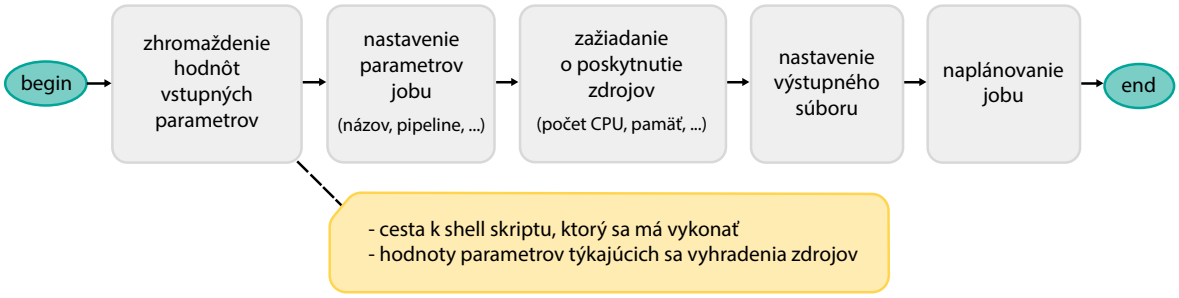
MPI

Ďalšiu možnosť predstavuje štandard pre odovzdávanie správ, rozhranie MPI, ktoré po nakonfigurovaní umožňuje komunikáciu s Grid enginom. Jedná sa o knižnicu, ktorá sprostredkúva komunikáciu technikou odovzdávania správ. Je to flexibilný, presný a efektívny štandard určený na písanie programov na platformách HPC, ktoré môžu bežať plne paralelne.[16] Umožňuje spúšťať joby v dvoch režimoch: [17]

- **single node** - job beží v jednom node (teda sériovo)
- **multi node** - využíva sa tzv. paralelné prostredie, pričom joby bežia v dvoch alebo viacerých uzloch (tzv. nodes)

Z hľadiska poskytovanej funkčnosti sa jedná o jeden z najpokročilejších spôsobov spúšťania a plánovania jobov, pričom použitie rozhrania MPI možno prirovnať k úrovni používania príkazov príkazoveho riadku Grid enginu. Aj napriek tomu však nie je veľmi využívaná pri tvorbe wrapperov.

Poskytuje možnosť manažmentu jobov - ich zaradenie do fronty a rušenie, prideľovanie zdrojov jobom pri plánovaní a zmenu nastavení fronty. Oproti DRMAA má ešte jednu výhodu, ktorú predstavuje podpora interaktívneho režimu a možnosť použitia paralelného prostredia.



Obr. 2.5: Postup zaradenia jobu do fronty na vykonanie pomocou knižnice MPI

Štandard MPI možno priamo využívať prostredníctvom jazyka bash alebo postredníctvom knižnice, ktorá je dostupná v jazykoch C, C++ a Fortran. Existuje však aj alternatíva pre jazyk Python. Jedná sa o knižnicu mpi4py, ktorá dosahuje vysokej rýchlosti komunikácie vďaka PBI aj napriek tomu, že sa jedná o interpretovaný jazyk.

2.3 Generovanie formulárov

Keďže sa počty a typy vstupných parametrov plánovaných skriptov a ich hodnoty môžu líšiť, bolo potrebné nájsť spôsob, ako generovať formuláre, pomocou ktorých by užívateľ mohol hodnoty parametrov zadať. Z tohto dôvodu bol vykonaný prieskum balíčkov poskytujúcich túto funcionalitu. Hlavnou požiadavkou bolo, aby balíčky dokázali z JSON objektu vygenerovať formulár, prípadne i validovať užívateľom zadané hodnoty. Keďže klientská časť aplikácie bude realizovaná formou Node.js aplikácie s využitím knižnice React, prieskum je zameraný primárne na balíčky, ktoré sú vytvorené prostredníctvom tejto knižnice. Všetky uvedené balíčky sú voľne dostupné na stiahnutie na stránkach NPM.

2.3.1 JSONForms

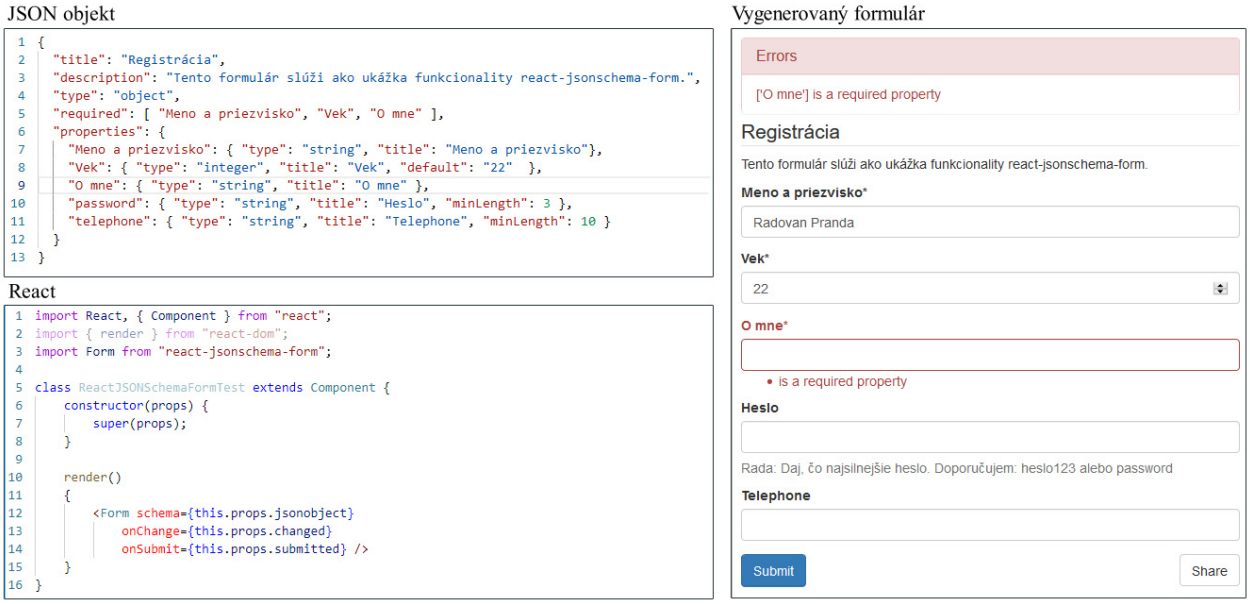
JSONForms je pravidelne rozširovaný a udržiavaný balíček, ktorý umožňuje generovanie formulárov z JSON objektov. Jednotlivé atribúty je možné deliť vrámci formulára do kategórii a tiež určiť, či sa je parameter povinný. Balíček sa tiež stará o automatickú validáciu vstupov na základe prednastaveného typu vstupu. Obsahuje veľké množstvo podporovaných typov vstupov, medzi ktoré patria okrem textových, numerických a boolovských vstupov aj dátumy vo forme kalendárov a polia, respektíve listy.[18][19]

Jedná sa teda o pomerne rozsiahly balíček so širokou paletou podporovaných dátových typov. Balíček je dostupný pre React a Redux, ale aj pre Angular.js. Pričom niektoré funkcie, ktoré sú dostupné vo verzii pre AngularJS, nie sú dostupné v React, Redux verzii a naopak.

2.3.2 react-jsonschema-form

Jedná sa o róbusnejší balíček, svojou funkčnosťou podobný JSONForms, ktorý je pravidelne udržiavaný a rozširovaný. [20] Umožňuje pokročilé generovanie formulárov z JSON objektov. Okrem funkcionalít, ktoré poskytuje JSONForms, navyše poskytuje pridávanie textových odstavcov do formulárov, nastavenie témy a farieb daného formulára. Na rozdiel od predchádzajúceho balíčka, umožňuje tiež nastaviť url a funkciu, ktorá má byť pri zaslaní dát použitá.

Veľkou výhodou je možnosť písania vlastnej validácie, kedy kontrola hodnoty prebieha automatizovane vďaka funkcii regex. Naopak nevýhodou je omnoho väčšia veľkosť, avšak vzhľadom na to, že sa veľkosti balíčkov pohybujú v rádoch jednotiek, nanajvýš desiatok megabajtov, je táto nevýhoda relatívne zanedbateľná. Ďalšou nevýhodou je relatívne zložitá a neobvyklá štruktúra objektu



Obr. 2.6: Ukážka generovania formulárov pomocou balíčka react-jsonschema-form

obzvlášť v prípade zanorovania parametrov do podformulárov. Neobvyklá je aj realizácia nastavenia parametra ako povinného, kedy sa povinnosť daného parametra neurčuje vrámci daného parametra, resp. atribútu, ale vrámci formulára, respektíve jeho sekcie v prípade zanorovania.

Dáta možno z komponenty formulára získať vo forme objektu prostredníctvom metódy onChange. Pričom platí, že objekt kopíruje štruktúru formulára a zachováva zanorenie atribútov ako aj ich hodnôt. Nezachováva však povinné parametre, ktoré v prípade, že ich užívateľ nevyplní, nie sú obsiahnuté v získanom objekte. Ako identifikátor atribútu sa používa vždy hodnota priradená atribútu title. Na obrázku 3.13 vidieť príklad generovania jednoduchého formulára. Pri bližšom náhľade na JSON objekt si možno všimnúť, že balíček umožňuje i vkladanie popiskov formulára a nápovedy pre jednotlivé atribúty.

2.3.3 Winterfell

Posledným rozoberaným balíčkom je knižnica Winterfell, ktorá nepatrí medzi najrozsiahlejšie avšak ani medzi najmenšie. Poskytuje široké spektrum funkcii, medzi ktoré patrí okrem iného aj možnosť písania vlastnej validácie, teda aj možnosť nastavenia vlastných chybových hlášok, aplikácie vlastného štýlovania a automatizovaná validácia jednotlivých dátových typov vstupov. Oproti ostatným balíčkom vyniká možnosťou zadefinovania vlastných typov vstupov a ľahkého rozšírenia.

Užívateľovi umožňuje tiež deaktivovať tlačidlá použité vo formulári. V súvislosti s akciami umožňuje balíček tvorbu eventov. Prostredníctvom props však poskytuje možnosť zaregistrovať len 4 typy eventov. Jedná sa o akcie, ktoré sa spustia pri:

- vyrendrovaní formulára

- zmene niektorej z častí formulára
- zmene aktívneho panelu
- submitovaní formulára

Samozrejmosťou je možnosť nastavenia prednastavenej hodnoty. Knižnica tiež umožňuje vykonávať podmienené submitovanie formulára. Na rozdiel od vyššie spomenutých knižníc neobsahuje niektoré pokročilejšie prvky, medzi ktoré patrí napr. vstup prostredníctvom kalendára. Ďalšou nevýhodou je relatívne zložitá štrutúra vstupného objektu, ktorý môže mať rekurzívnu štruktúru.[21]

2.3.4 Porovnanie balíčkov

Väčšina testovaných knižníc poskytovala rovnaké funkcie a komponenty. Hlavné rozdiely boli v štruktúre objektu, ktorý pri zmene niektorej z hodnôt vracala hlavná rodičovská komponenta obvykle prostredníctvom metódy onChange. V tomto smere boli zaznamenané počas testovania dva prístupy, respektíve spôsoby konštrukcie objektu. Metóda počas volania vracala:

- **lineárneárne štrukturovaný objekt**

  V tomto prípade sa štruktúra podobá slovníku, nakoľko ho tvoria dvojice kľúč - hodnota. Atribúty sú identifikátormi (unikátnymi kľúčmi) jednotlivých užívateľom zadaných hodnôt. Typy hodnôt pritom zodpovedajú typom definovaným vo vstupnom JSON objekte. Vo všetkých prípadoch sa pritom jednalo o hodnoty primitívneho dátového typu, prípadne polia.

- **stromovú štruktúru objektov**

  Výstupom bol objekt kopírujúci štruktúru vstupného JSON objektu. Jednalo sa teda o štruktúru podobnú slovníku s rekurzívnou štruktúrou, kde kľúčmi sú buď identifikátory alebo mená atribútov, podsekcii či podformulárov. Priradenou hodnotou pritom bývajú hodnoty alebo polia primitívnych typov, prípadne ďalší objekt podobný vyššie spomínanému rekurzívnemu slovníku. Stupeň rekurzie a poloha atribútu pritom zodpovedá jeho polohe a stupňu vrámci vstupného objektu typu JSON.

Počas prieskumu a testovania sa zistilo, že väčšina balíčkov využíva na štýlovanie knižnicu reactstrap, čo je ekvivalent Bootstrapu pre Node.js aplikácie. Veľkosti knižníc sa pritom pohybovali v rádoch jednotiek nanajvýš desiatok megabajtov. Z hľadiska poskytovaných funkcionalít vynikal balíček react-jsonschema-form. Jednalo sa o najväčší a najróbusnejší balíček. Presným opakom bol balíček Winterfell, ktorý poskytoval najmenej funkcionalít a možností. Do tohto porovnania nebola zahrnutá knižnica vytvorená pre potreby výslednej aplikácie, nakoľko je jej venovaná podkapitola 3.5, ktorá obsahuje aj porovnanie s preskúmanými a otestovanými balíčkami.

Kapitola 3

Vývoj aplikácie

Tretia kapitola sa zaoberá samotným návrhom a vývojom výslednej aplikácie. Vrámci jednotlivých podkapitol bude rozoberaný vývoj klienta a API, s ktorým klient komunikuje, ako aj problémy, ktoré počas vývoja nastali. Okrem toho je súčasťou popis vývoja knižnice umožňujúcej generovanie formulárov z objektu typu JSON, ktorá bola vytvorená namieru na základe požiadavok vytváranej aplikácie a následné porovnanie s knižnicami popísanými v podkapitole 2.3.

3.1 Požiadavky

Ešte pred samotným návrhom a vývojom výslednej aplikácie je potrebné ujasniť, aké funkcie by mala výsledná aplikácia poskytovať. Keďže je cieľom vytvoriť webovú aplikáciu, ktorá bude poskytovať užívateľovi čo najväčšiu možnú kontrolu nad Grid enginom, medzi hlavné požiadavky patrí, aby aplikácia umožňovala:

- plánovať nové a spravovať bežiace pipeline tvorené jobmi s rôznymi nárokmi na zdroje zariadenia
- sledovať aktuálny stav a priebeh jobov a pipeline
- nahrávať, upravovať a spravovať nahrané skripty
- spájať na seba naväzujúce skripty s rôznymi požiadavkami na zdroje do pipeline
- sledovať vyťaženie hostov, spotrebované a dostupné zdroje

Aby užívateľ, prípadne správca systému v prípade nastania chyby vedel, kde nastala chyba pri vykonávaní pipeline či v behu enginu, API alebo v niektorom z jobov tvoriacich pipeline, je potrebné do aplikácie pridať aj funkciu zobrazenia logu enginu a jednotlivých hostov, ktoré prepája prostredníctvom Grid enginu zariadenie s master daemonom.

3.2 Užívateľské role

Funkcie budú vrámci aplikácie rozdelené medzi užívateľské role:

- **Neautorizovaný užívateľ** - aktér nadobúda túto rolu od úplného začiatku až do doby pokiaľ sa úspešne neprihlási. Okrem prihlásenia nemá prístup k žiadnej z funkcii aplikácie.

- **Autorizovaný užívateľ** - užívateľ, ktorý sa v takejto role nachádza, prešiel úspešne prihlásením a môže pristupovať k jednotlivým funkciám aplikácie. Ich množstvo je pritom závislé na miere práv, ktoré mu boli pridelené. Na základe pridelených práv možno rozlišovať role:

  - **Užívateľ** - sú mu sprístupnené základné funkcie aplikácie, akými sú vytváranie, úprava, publikovanie a mazanie skriptov, ich spájanie do pipeline template. Ďalej tiež spúšťanie pipeline na zariadeniach, ku ktorým mu bol udelený prístup, sťahovanie výsledkov, prezeranie ich logu a monitorovanie ich priebehu.

  - **Administrátor** - je rola s najväčšími právami a prístupom k funkciám. Okrem tých sprístupnených už roli užívateľ, je mu umožnené konfigurovať systém. Môže pridávať, editovať a mazať výpočtové zariadenia, interprétery a kompilátory ako aj užívateľov, ktorým môže okrem iného odoberať a udeľovať administrátorské práva či prístup k aplikácii. Mimo toho je mu umožňené spúšťať, editovať, mazať skripty a pipeline template všetkých užívateľov bez ohľadu na nastavenie viditeľnosti a pridelených právach prostredníctvom zdieľania. Má tiež kontrolu nad všetkými bežiacimi pipeline, ktoré môže kedykoľvek zastaviť.

Okrem vyššie spomenutých rolí, ktoré budú fungovať vrámci výslednej aplikácie, bude mať výrazný vplyv na mieru poskytovanej funkčnosti rola, ktorá je vrámci enginu a operačného systému pridelená účtu, pomocou ktorého prebieha komunikácia s Grid enginom. V prípade, že sú účtu uvedenému v konfigurácii aplikácie vrámci enginu pridelené práva užívateľa, budú mať užívatelia aplikácie slabé práva vrámci Grid enginu a tým veľmi obmedzené možnosti aj vrámci aplikácie. Najviac ovplyvnené bude pritom práve plánovanie a správa pipeline. Nakoľko, ako sa uvádza už v podkapitole 2.1.3, rola užívateľa má vrámci enginu najmenšie práva, umožňuje vidieť a spravovať iba svoje joby, a neumožňuje upravovať konfiguráciu ani inak výrazne zasahovať do nastavení.

Naopak, keby mal účet uvedený v konfigurácii aplikácie pridelenú vrámci enginu rolu manažéra alebo operátora, užívateľ by mohol mať nad enginom totálnu kontrolu a ovládať všetky jeho časti - vrátane vykonávania zmien v konfigurácii za behu, pozastavovania a spúšťania front. Táto závislosť má veľký význam z pohľadu bezpečnosti, nakoľko umožňuje regulovať veľkosť dopadu konania užívateľov aplikácie na celkové správanie enginu.

3.3 Voľba vývojového nástroja

Vzhľadom na to, že sa jedná o webovú aplikáciu bolo zvolené spomedzi vývojových nájstrojov .NET Core Web API pre backend aplikácie. Hlavným dôvodom tohto výberu je vysoká rýchlosť, miera kompatibility a možnosti jednoduchého rozšírenia o ďalšie funkcionality. Ďalšou výhodou je, že .NET Core je schopné bežať ako na serveroch s OS Windows, tak na serveroch s OS Linux a prináša možnosti v prípade neskoršej potreby vytvorenia desktopovej či mobilnej aplikácie na sledovanie a riadenie pipeline a jobov spúšťaných prostredníctvom Grid enginu.

V prípade klientskej časti bolo na výber viacero možností. Jednou z možností bolo využitie MVC, ktoré v tomto prípade úplne nevyhovovalo. Hlavným dôvodom bolo o niečo obtiažnejšie generovanie formulárov z JSON objektov. Medzi ďalšie možnosti patrilo využitie niektorého z JavaScriptových prostredí, akými sú Angular či zvolené Node.js, ktorého hlavnou výhodou je veľké množstvo dostupných balíčkov, jednoduchý vývoj a ľahká rozšíriteľnosť. Pričom spomedzi balíčkov bola vybraná za základ knižnica React.[22]

3.4 API

Nasledujúca podkapitola bude venovaná vývoju API, ktoré komunikuje s Grid enginom a PostgreSQL databázou, ktorá slúži na uloženie informácii o užívateľoch, nahraných skriptoch, pipeline templatoch ako aj práve prebiehajúcich pipelinách. Väčšina informácii je ukladaná vrámci databáze, pričom na mapovanie objektov sa využíva Entity Framework. Objektovo-relačné mapovanie a komunikácia prebieha konkrétne prostredníctvom knižnice Npgsql.EntityFrameworkCore, ktorá bola k aplikácii pripojená prostredníctvom NuGet.

Samotné API sa skladá z kontrolérov, ktoré možno rozdeliť do dvoch kategórii. Kontroléry poskytujúce základné informácie o položkách uložených v databáze, u ktorých sa jedná o praktické využitie návrhového vzoru LazyLoad, na zobrazenie najzákladnejších informácii o zariadeniach, skriptoch, atď. Ide napríklad o kontroléry: DevicesController, HostsController, InterpretersController, PipelinesController a podobne. Ku každému z nich existuje kontrolér poskytujúci detailnejšie informácie o jednotlivých položkách a podieľajúci sa na niektorej z hlavných funkcionalít aplikácie.

Autorizácia

Vzhľadom na to, že na sprístupnenie hlavnej funkčnosti aplikácie je potrebná autorizácia, bol vytvorený samostatný kontrolér s názvom AuthController. Na prihlásenie sa využíva metóda jednoduchého prihlásenia, Basic Auth. V prípade úspešného prihlásenia sa užívateľovi vráti bearer token spolu s dátumom a časom jeho expirácie, ktorý slúži na sprístupenie hlavných funkcii aplikácie.

Správa skriptov

Jednou z hlavných funkcii poskytovaných aplikáciou je spúšťanie skriptov, je preto potrebné, aby bolo možné súbory so skriptami nahrať na server, kde budú môcť byť uložené a v prípade potreby nahrané na server s Grid enginom, kde bude naplánované ich vykonanie. Možnosť nahrať na server s API nové skripty či upravovať už uložené skripty poskytuje ScriptController tak, že sa na strane klienta vykoná načítanie súboru najskôr do pamäte prehliadača, kde môže užívateľ skript editovať a následne sa uložením vykoná nahranie súboru vo forme BLOB na server s API. Skriptu možno prideliť vlastný názov a popis, prekladač, ktorým bude spúšťaný, nastaviť a editovať jeho vstupné parametre a požiadavky na zdroje, ktoré budú požadované od hosta prostredníctvom Grid enginu. Užívateľ môže tiež nastaviť skript ako verejný alebo súkromný, rovnako ako aj zdieľať skripty s ostatnými užívateľmi, pričom miera s akou bude môcť užívateľ, s ktorým je skript zdieľaný manipulovať so skriptom je závislá na tom, aké práva mu jeho majiteľ pridelí. Výnimku v tomto smere predstavuje administrátor, ktorému je udelená úplna kontrola nad všetkými skriptami nahranými do systému bez ohľadu na ich viditeľnosť alebo zdieľanie. Vďaka tomu môže administrátor v prípade potreby vykonať editáciu skriptu či zmazať skript - napr. v prípade, že by porušoval podmienky stanovené správcom systému.

Súbory s kódom sú vrámci servera s API ukladané do lineárnej štruktúry bez prípony, aby nebolo nutné v prípade zmeny prekladača užívateľom vykonávať prepis na disku. Každému je pridelené vrámci databáze unikátne id, ktoré sa používa ako názov súboru uloženého na disku.

Správa pipeline a jobov

Vrámci správy sa delia joby na dva typy:

- **Manažované** - sú joby tvoriace pipeline, ktorú užívateľ vytvoril prostredníctvom aplikácie, ako je popísané v kapitole 3.4.2. Na vytváraní takýchto jobov sa podieľa JobController, ktorý tak predstavuje jeden z prvých článkov v životnom cykle pipeline, pretože vrámci neho prebieha generovanie zavádzacích skriptov pipeline a príprava presunu dát na HPC.

- **Nemanažované** - jedná sa o joby, ktoré si užívateľ naplánoval bez použitia aplikácie. Napríklad tak, že sa prihlásil prostredníctvom SSH k danému zariadeniu, na ktorom vykonal qsub vlastného skriptu. Sledovanie a správu takýchto jobov umožňuje UnhandledJobController.

Okrem plánovania a správy pipeline poskytuje detailné informácie o jednotlivých joboch, ich aktuálnom stave, výpis logu, ako aj postupnosti skriptov, ktoré ich tvoria, parametre, s ktorými boli spúšťané, ako aj zdroje, ktoré sú nimi požadované. O synchronizácii dát týkajúcich sa jobov a pipeline bude pojednávať podkapitola 3.4.5.

Správa pipeline templatov

Aby užívateľ nemusel v prípade opakovaného spúšťania pipelinu neustále vytvárať nanovo, PipelineController umožňuje užívateľovi vytvárať a spravovať predchystané pipeliny. Užívateľ si teda vrámci aplikácie môže svoj plán zložený z postupnosti viacerých skriptov, ktoré na seba istým spôsobom naväzujú, uložiť a následne kedykoľvek opätovne nahrať z databázy, a spustiť s rovnakým alebo iným nastavením hodnôt vstupných parametrov. Jednotlivé plány vykonávania skriptov sú ukladané do databázy vrátane objektov JSON, ktoré obsahujú preddefinované alebo predvyplnené hodnoty vstupov. V prípade veľkého počtu parametrov, teda nie je potreba zadávať parametre vždy nanovo, stačí ich jednoducho načítať s templatom z databázy. Na celkovej funkčnosti tohto controlleru sa z veľkej časti na strane klienta podieľa rovnako ako v prípade ScriptController-a knižnica umožňujúca generovanie formulárov, ktorej je venovaná podkapitola 3.5.

Správa prekladačov

Vzhľadom na to, že sú ukladané na disku súbory bez prípon, je potrebné im príponu priradiť počas ich prípravy na vykonanie. InterpreterController slúži na správu prekladačov, formátu ich zavádzacích príkazov, ako aj súborových prípon, ktoré budú skriptom priradené tesne pred prenosom. Prekladače a ich detaily môže spravovať iba administrátor, ich zoznam je však dostupný do istej miery aj užívateľom, napríklad pri pridávaní a editácii skriptu.

Správa zariadení, hostov a front

Medzi neodmysliteľné časti aplikácie patrí možnosť pridať zariadenie, na ktorom môže byť pipeline naplánovaná. Zariadenia pritom vrámci systému delíme na dva typy:

- **Master host** - zariadenie, na ktorom je umiestnený master daemon a na ktorom prebieha naplánovanie pipeline. Pridávať a editovať ich môže administrátor prostredníctvom DeviceControllera. Ďalej ho možno v prípade potreby deaktivovať, napr. z dôvodu údržby, či otestovať pripojenie k zariadeniu, v prípade, že komunikácia prebieha prostredníctvom SSH, prípadne aj SFTP. Aby správca systému nemusel manuálne pripravovať pracovné prostredie na strane master hosta, umožňuje ho tento kontrolér automatizovane vytvoriť.
- **Execution host** - zariadenie, na ktorom sa nachádza execution daemon a je spravované zariadením, na ktorom beží master daemon. Užívateľ môže iba sledovať ich aktuálny stav a vyťaženie či to, či je host ešte stále dostupný a k akým frontám má daný host prístup. Detailné informácie týkajúce sa hostov v tomto smere poskytuje HostController.

Pričom platí, že jeden master host môže riadiť a spravovať viacerých execution hostov. Každému z execution hostov sú pridelené fronty, ku ktorým má prístup. Každá z front má pritom definované obmedzenia, ktoré definujú množstvo a typ zdrojov, ktoré si naplánovaný job môže vyžiadať od

hosta, ako aj počet slotov, teda maximálny počet jobov, ktoré môžu bežať paralelne. Užívateľ môže tieto obmedzenia rovnako ako stav slotov danej fronty sledovať prostredníctvom QueueControllera.

Správa užívateľov

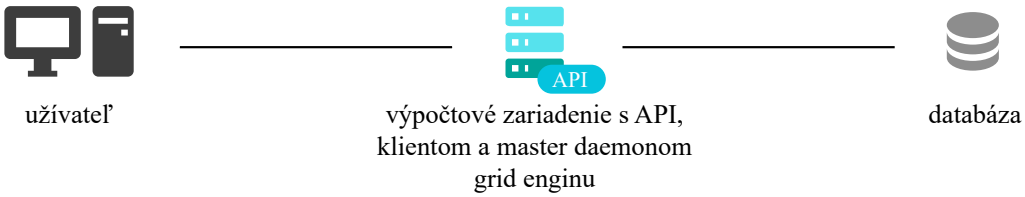
S manažmentom aplikácie úzko súvisí aj správa a vytváranie užívateľov. Všetky metódy a funkcie poskytované týmito kontolérmi sú prístupné exkluzívne administrátom. Prostredníctvom UserControllera môže administrátor meniť heslá či administrátorské práva úžívateľov, ale aj vytvárať uživateľov nových, či užívateľov blokovať v prípade, že porušujú pravidlá.

3.4.1 Komunikácia API s Grid enginom

Ako vyplýva z predchádzajúcich kapitol, hlavnú funkčnosť aplikácie zabezpečuje API vďaka spolupráci s Grid enginom. Komunikácia prebieha vždy s master hostom. Pre jednoduchosť bude preto vrámci schém uvádzané iba zariadenie s master daemonom. Na toto zariadenie môže byť ďalej napojených viacero execution hostov. API poskytuje v tomto smere tri režimy, respektíve módy komunikácie so zariadením v závislosti na umiestnení voči master hostovi:

- **Lokálny režim**

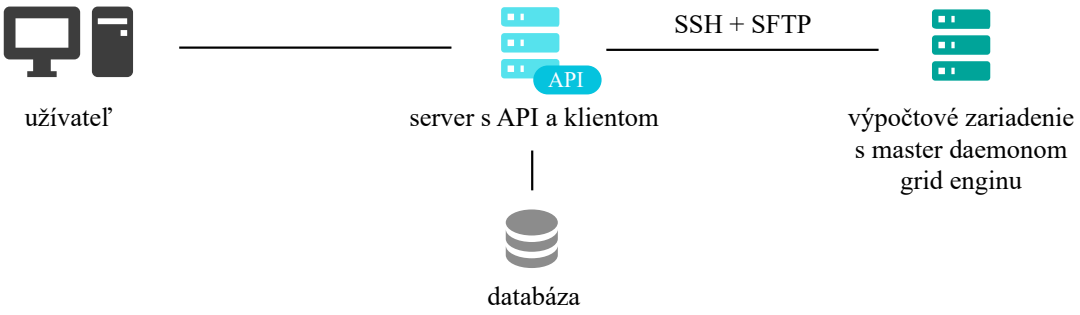
  V prípade, že API beží na zariadení, na ktorom beží aj master daemon Grid enginu, komunikácia prebieha prostredníctvom knižnice System.Diagnostics.Process, ktorá umožňuje zasielanie príkazov priamo do prostredia terminálu a na prenosy dát sa používa knižnica System.IO. Príkladom takého zariadenia je zariadenie na obrázku 3.1.



Obr. 3.1: Schéma komunikácie u lokálneho režimu

- **Režim vdialeného prístupu**

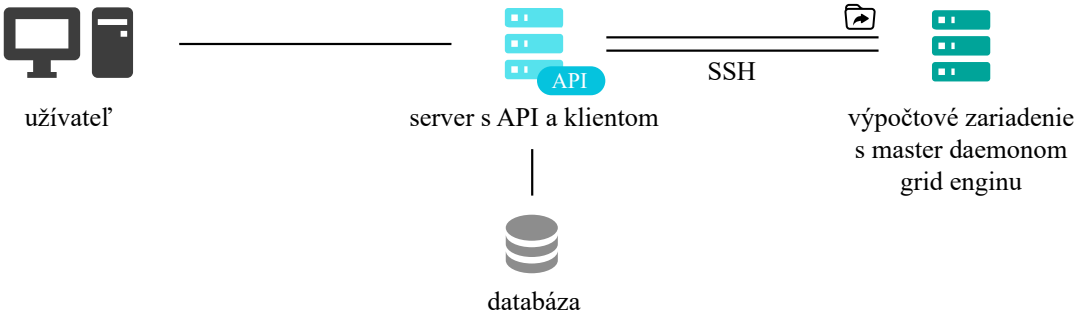
  Keďže v prípade vzdialeného ovládania sa API a master daemon nenachádzajú na tom istom zariadení, komunikácia prebieha prostredníctvom príkazov zasielaných do prostredia terminálu cez SSH a prenos dát prebieha prostredníctvom SFTP. Z čoho vyplýva, že na strane API je vyžadovaná prítomnosť IP adresy a portu výpočtového zariadenia s enginom, na ktorom bude prebiehať naplánovanie, ako aj prihlasovacích údajov užívateľa, prostredníctvom ktorého prebieha komunikácia, čo je hlavnou nevýhodou a zároveň bezpečnostnou slabinou tohto režimu, nakoľko údaje sú ukladané vrámci databázy.



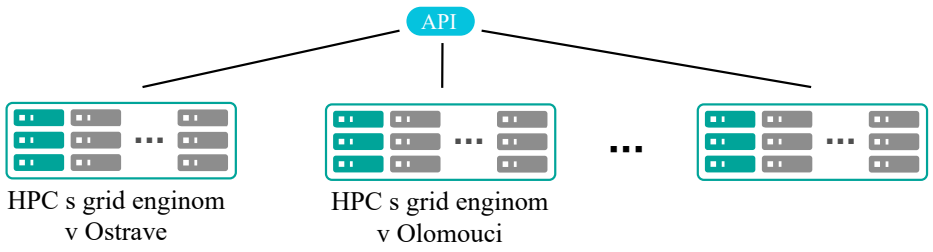
Obr. 3.2: Schéma komunikácie u režimu vzdialeného prístupu

- **Režim vzdialeného prístupu so sieťovou zložkou**

  Posledný režim predstavuje akýsi medzistupeň medzi vyššie uvedenými režimmi. Rozdiel spočíva v tom, že prenos dát nie je vykonávaný prostredníctvom SFTP, ale knižnicou System.IO. Dáta sú presúvané do zložky, ktorá je zdieľaná výpočtovým zariadením so zariadením s API cez sieť. Samotné naplánovanie pipeline následne prebieha rovnako ako v prípade režimu vzdialeného prístupu cez SSH.



Obr. 3.3: Schéma komunikácie u režimu vzdialeného prístupu so sieťovou zložkou



Obr. 3.4: Príklad - schéma komunikácie API s navzájom nezávislými Grid enginmi

Na záver je nutné podotknúť, že API je stavané tak, aby mohol pomocou neho užívateľ pristupovať k viacerým master hostom, čo umožňuje pristupovať k viacerým HPC naraz bez toho, aby boli navzájom prepojené prostredníctvom Grid engine - viď obrázok 3.4.

3.4.2 Zostavenie pipeline

Ešte pred popisom samotného zostavenia pipeline, ktorá je následne zaslaná a naplánovaná na výpočtovom zariadení s Grid enginom, je potrebné vymenovať a zadefinovať stavy, v ktorých sa môže nachádzať. Na ukladanie aktuálneho stavu sa využíva databáza, pričom o aktuálnosť stavu a informácii sa stará trojica služieb, ktoré budú popísané v nasledujúcich podkapitolách popisujúcich životný cyklus pipeline.

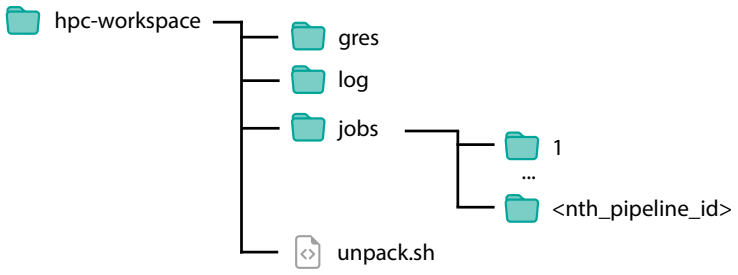
Pipeline, ktorá bola úspešne zaslaná API a prešla validáciou na strane klienta, môže nadobúdať stavy: Naplánované, Čakanie na prenos na výpočtový server, Prenos dát dokončený, Zaradené do fronty, Prebieha výpočet, Výpočet dokončený, Príprava výsledkov, Prenos výsledkov z výpočtového servera, Pipeline dokončená, Chyba, Zrušená alebo Stav neznámy.

Prerekvizity

Aby proces na strane výpočtového zariadenia prebehol úspešne, je potrebné prostredie, v ktorom aplikácia na jeho strane pracuje, pripraviť. Pracovný adresár možno automatizovane vytvoriť prostredníctvom kontroléra DeviceController, ako bolo spomenuté už v úvode tejto kapitoly. Stromovú štruktúru pracovného adresára aplikácie umiestneného na strane výpočtového zariadenia s master daemonom, môžno vidieť na obrázku 3.5.

Koreňom prostredia je adresár hpc-workspace, v ktorom sú umiestnené adresáre:

- **gres** - obsahuje skomprimované výsledky dokončených pipeline, ktoré čakajú na prenos
- **log** - obsahuje súbory s príponou log (presmerovaný výpis z terminálu) a temp (zmeny stavu s identifikátormi jobov, ktoré ju tvoria), pričom každej pipeline je pridelený vlastný
- **jobs** - obsahuje pracovné adresáre práve vykonávaných alebo čakajúcich pipeline



Obr. 3.5: Pracovný priestor aplikácie na strane výpočtového zariadenia

Okrem adresárov je tu tiež umiestnený súbor unpack, ktorý slúži ako zavádzací súbor každej pipeline. Vykoná extrakciu dát prenesených zo serveru s API na výpočtový server a vytvorí pracovné prostredie pipeline, ktoré bude popísané nižšie.

Celý proces plánovania začína už na strane klientskej aplikácie, ktorá umožňuje užívateľovi nahrať kód skriptu, zadefinovať potrebné parametre a zostaviť následne pipeline, ktorá sa bude vykonávať. Proces vykonávania možno potom zjednodušene popísať nasledujúcimi piatimi hlavnými krokmi:

Definovanie parametrov pipeline

Proces definovania parametrov začína v klientskej aplikácii v sekcii New pipeline, kde si užívateľ môže vytvoriť novú pipeline. Užívateľ musí zadať názov pipeline, ktorý je určený iba pre užívateľa na jednoduchšiu identifikáciu, ďalej zvoliť skripty, prípadne už uložený pipeline template a zadať hodnoty požadovaných parametrov. Dôležité je tiež, aby užívateľ zvolil zariadenie, ktoré pipeline vykoná.

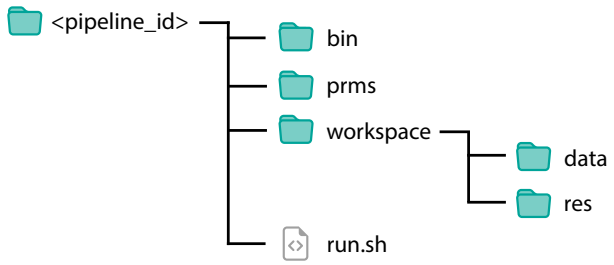
Príprava dát na prenos na výpočtový server

Dáta sú následne klientom zaslané API a spracované vrámci kontroléra JobController. V prípade, že pipeline má pridelené zariadenie, na ktorom sa naplánuje jej vykonávanie a obsahuje aspoň jeden skript, v databáze sa vytvorí záznam. Hodnota premennej reprezentujúcej posledný stav sa nastaví na 0, čo reprezentuje počiatočný stav - Naplánované a vykoná sa uloženie času vykonania tejto operácie.

Súbory skriptov, ktoré sa nachádzajú na serveri a nie sú označené ako virtuálne alebo fyzicky zmazané, sú pridané do súboru typu ZIP. V prípade parametrov sa vykoná zápis JSON objektov do súborov s príponou JSON, ktoré sú rovnako ako skripty umiestnené do spomínaného ZIP súboru. Okrem komprimácie vyššie spomínaných položiek, prebieha vytvorenie súboru run.sh, ktorý je do archívu pridaný na záver. Súbor sa skladá z dvoch hlavných častí: hlavná pipeline a metóda fin. Pričom metóda fin vykoná skomprimovanie adresára s výsledkami a zmazanie pracovného prostredia pipeline na disku. Hlavná pipeline definuje stratégiu, s akou budú jednotlivé skripty tvoriace pipeline naplánované. Na záver sa nastaví stav pipeline na Čakanie na prenos na výpočtový server. Jednotlivé zavádzacie príkazy skriptov sú zostavené využitím atribútu format z tabuľky Interpreters.

3.4.3 Prenos dát na výpočtový server a naplánovanie pipeline

Skomprimované dáta sú spracované následne službou DataTransferService, ktorá beží nepretržite na pozadí API. Okrem prenosu sa služba stará aj o extrahovanie obsahu archívu na strane výpočtového zariadenia, prípravu pracovného prostredia pipeline a naplánovanie. Využíva pritom jednoduchý skript s názvom unpack.sh napísaný v jazyku Bash nachádzajúci sa v pracovnom prostredí aplikácie. Ten sa na výpočtovom serveri vytvorí počas prípravy prostredia, ktorú je potrebné vykonať počas



Obr. 3.6: Pracovný priestor pipeline

konfigurovania nového zariadenia, prípadne kedykoľvek počas úpravy výpočtového zariadenia. Je nutné podotknúť, že bez vytvorenia pracovného prostredia plánovanie neprebehne a skončí chybou, nakoľko služba netestuje integritu predpripraveného prostredia. V prípade úspešného priebehu sa stav pipeline zmení na Prenos dát dokončený.

Ako bolo spomenuté v predchádzajúcom kroku, vykoná sa extrahovanie súboru ZIP, obsahujúceho všetky potrebné skripty a vstupné parametre. Všetky súbory sú extrahované do priečinku jobs/<unikátne id pipeline> umiestneného v pracovnom adresári aplikácie vytvorenom na výpočtovom serveri. Následne sa vytvoria adresáre pracovného prostredia pipeline, ako možno vidieť na obrázku 3.6. Pričom každý adresár má svoju vlastnú funkciu:

- **koreňový adresár (root)** - obsahuje bashový súbor run.sh, ktorý obsahuje postupnosť skriptov, z ktorých sa pipeline skladá
- **bin** - obsahuje skripty, ktoré budú spúšťané
- **prms** - obsahuje súbory typu JSON obsahujúce hodnoty parametrov zadané užívateľom
- **workspace** - pracovné prostredie skriptov, je určené na prácu a zmeny vykonávané vrámci behu jednotlivých skriptov
- **res** - adresár, do ktorého si môže užívateľ prostredníctvom svojich skriptov uložiť výsledné súbory, na konci sa potom obsah tohto adresára skomprimuje do archívu s príponou ZIP a prenesie na server s API, odkiaľ môže užívateľ svoje výsledky kedykoľvek stiahnuť

Po vytvorení prostredia nasleduje samotné naplánovanie pipeline. Realizuje sa príkazom qsub, o ktorom sa pojednávalo už v podkapitole 2.2.1. Úvodnému jobu je pritom nastavený názov vo formáte <prefix>-<unikátne id pipeline>-1-g. Predvolená hodnota prefixu je predvolene nastavená na "hpc". Pridelenie vlastného názvu jobom tvoriacim pipeline uľahčuje zastavenie pipeline v ktoromkoľvek bode jej vykonávania.

Pri naplánovaní každeho z krokov tvoriacich pipeline sa do súboru s príponou temp uložia informácie o joboch tvoriacich pipeline. Jedná sa o súbor obsahujúci dáta serializovné vo forme JSON objektov, pričom každy obsahuje:

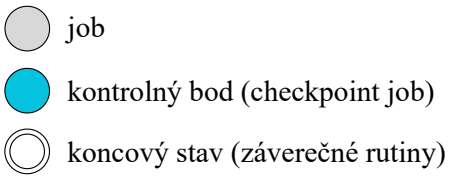
- **číslo kroku** - poradie skriptu vrámci pipeline
- **jobID** - prideľuje Grid engine jobu pri naplánovaní
- **príznak** - na základe ktorého API určuje, či sa jedná o krok alebo checkpoint (kontrolný bod) vrámci pipeline
  - **krokov pipeline** - príznak nie je nastavený
  - **kontrolný bod** - príznak je nastavený na hodnotu false
  - **záverečná rutina** - príznak je nastavený na hodnotu true

Po naplánovaní sa zmení hodnota stavu na Zaradené do fronty a zodpovednosť za synchronizáciu a získavanie aktuálneho stavu preberá služba GlobalQueueService. Tá kontroluje stavy jobov na zariadeniach v prípade, že sa job nachádza v jednom zo stavov zaradený do fronty, prebieha výpočet, výpočet dokončený, príprava výsledkov alebo stav neznámy, ktorý bude ďalej popísaný.

3.4.4 Stratégia použitá pri plánovaní

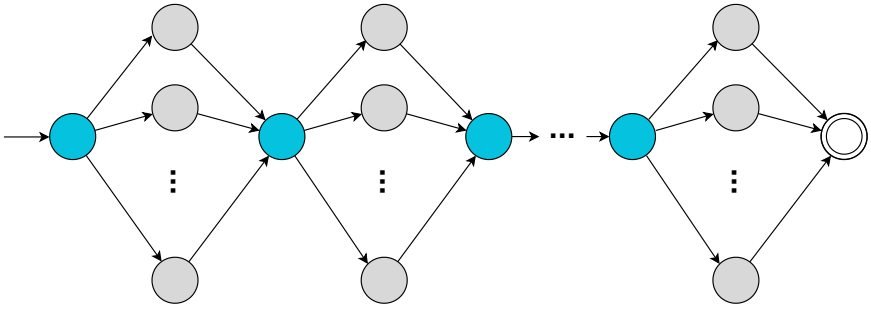
Počas vytvárania aplikácie boli zvažované viaceré stratégie, s akými plánovať pipeline prostredníctvom Grid enginu. Z týchto boli nakoniec zvolené 2 stratégie, ktoré budú vrámci tejto podkapitoly popísané a porovnávané. Pri plánovaní boli využité tri typy jobov, resp. uzlov (viď obrázok 3.7):

- **kontrolný bod (checkpoint job)** - jedná sa o uzol, vrámci ktorého prebieha naplánovanie jobov, prípadne aj čakanie na dokončenie jobov naplánovaných vrámci predchádzajúceho kontrolného bodu
- **job** - uzol, vrámci ktorého prebieha spustenie užívateľom vybraného skriptu, s požadovanými vstupnými parametrami, ktoré zadal prostredníctvom klienta a so zdrojmi, ktoré mu boli pridelené na základe požiadavok, ktoré zadal pri vytváraní skriptu
- **koncový stav (resp. záverečné rutiny)** - predstavuje v prípade oboch stratégii uzol, v ktorom prebieha volanie záverečných rutín, ktoré sú popísané v podkapitole 3.4.5.



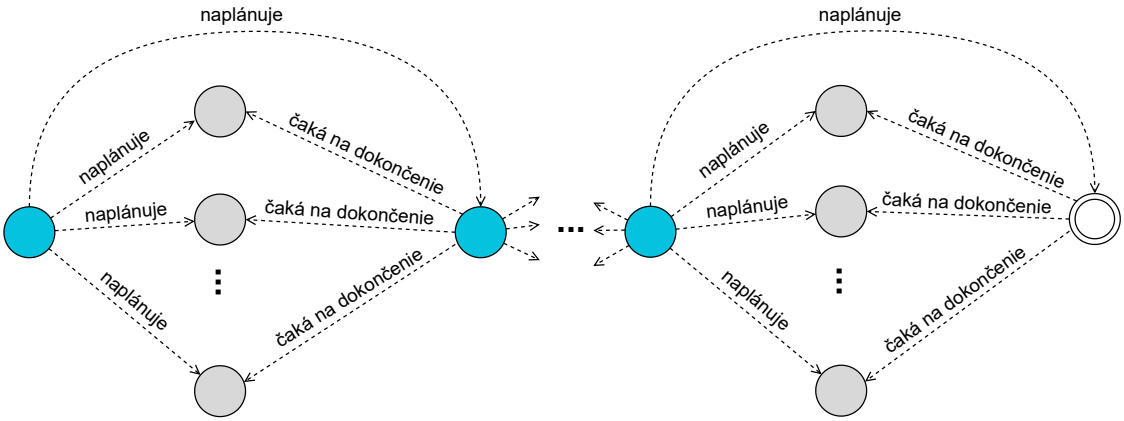
Obr. 3.7: Legenda stavov

Postupné plánovanie

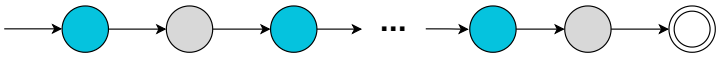


Obr. 3.8: Automat znázorňujúci vykonávanie pipeline s využitím postupného plánovania

Princíp fungovania prvej stratégie spočíva v zhlukovaní krokov pipeline do skupín, vrámci ktorých môžu joby bežať paralelne, pričom sa medzi takýmito skupinami vždy nachádza kontrolný bod čakajúci na dokončenie všetkých jobov z predchádzajúcej skupiny, viď obrázok 3.9. Po dokončení všetkých paralelne bežiacich jobov, nastane spustenie kontrolného uzla, ktorý naplánuje následujúcu skupinu jobov. Priebeh vykonávania môžno potom znázorniť automatom, ktorý je vidieť na obrázku 3.8, pričom pipeline prostredníctvom kontrolného bodu vždy čaká na dokončenie všetkých jobov, ktoré boli naplánované predchádzajúcim kontrolným bodom. Plánovanie jobov teda prebieha postupne a nevznikajú pri ňom žiadne vnútorné cykly. Hlavnou výhodou tejto stratégie je menšia réžia vrámci Grid enginu a výrazne menšie množstvo dát, ktoré je potrebné spracovať počas synchronizácie API s Grid enginom. Vďaka čomu budú užívateľovi zobrazované aktuálnejšie informácie.



Obr. 3.9: Vzťahy medzi uzlami vrámci pipeline naplánovanej postupne



Obr. 3.10: Ukážka najhoršieho prípadu u pipeline naplánovanej postupne

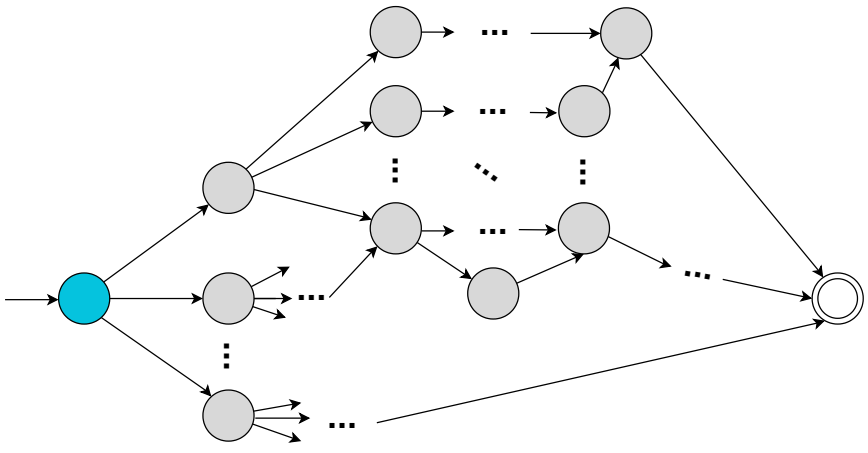
Nevýhodou tejto stratégie je veľká réžia spojená s plánovaním zo strany API v prípade rozsiahlych pipeline, u ktorých sa náväznosť jobov blíži sériovému prístupu - viď obrázok 3.10, kedy naplánovaná pipeline obsahuje zbytočne veľké množstvo kontrolných jobov, ktorých počet sa v najhoršom prípade blíži 2n+1, kde n je počet jobov tvoriacich pipeline.

Okamžité naplánovanie

Oproti predchádzajúcej stratégii sa pri okamžitom naplánovaní nachádzajú vrámci pipeline iba dva hlavné kontrolné body. Jedným je počiatočný stav, ktorý vykoná naplánovanie všetkých jobov tvoriacich pipeline naraz. Okamžite po naplánovaní je naplánovanému jobu prikázané čakať, dokiaľ sa plánovanie úplne nedokončí.

Druhým stabilným kontrolným bodom je koncový stav, počas ktorého sa vykonávajú záverečné rutiny. Ten čaká na dokončenie všetkých jobov tvoriacich pipeline. Medzi ostatnými uzlami existujú následne prechody a vzťahy tak, ako ich zadefinuje užívateľ, pričom užívateľovi nie je umožnené vytvárať medzi stavmi cykly. Dôvodom zavedenia tohto pravidla je garantovanie dosiahnutia koncového stavu. Vzhľadom na to, že plánovanie jobov prebieha naraz, už počas plánovania je potrebné poznať joby, na ktoré bude musieť čakať job vykonávajúci záverečné rutiny. Obecne teda platí, že každý z jobov tvoriacich pipeline môže predstavovať akýmsi spôsobom kontrolný bod, nakoľko čaká na dokončenie predchádzajúceho jobu, prípadne jobov.

V takomto prípade koncový uzol čaká na dokončenie každého jobu, ktorý je súčasťou naplánovanej pipeline. To má za následok omnoho väčšiu réžiu vrámci Grid enginu v prípadoch, kedy je pipeline priveľmi rozsiahla. Umožňuje však realizovať viaceré scenáre, ktoré by prostredníctvom predchádzajúcej stratégie nebolo možno realizovať, prípadne nebolo možné realizovať efektívne. Z toho dôvodu bol práve tento spôsob plánovania zvolený ako výsledné riešenie.



Obr. 3.11: Automat znázorňujúci vykonávanie pipeline s využitím okamžitého plánovania

3.4.5 Priebeh a záverečné rutiny

Sledovanie priebehu vykonávania pipeline prebieha na pozadí prostredníctvom služby GlobalQueueService. Služba nadväzuje spojenie so zariadeniami, na ktorých boli pipeline naplánované, prípadne bolo započaté ich vykonávanie. Synchronizácia pritom neprebieha u pipeline, ktoré boli zrušené užívateľom alebo ukončené chybou. Špeciálny prípad predstavujú v tomto prípade tie, u ktorých zlyhalo získanie aktuálneho stavu. Tu dochádza k rozhodovaniu, či sa má služba pokúsiť o opätovnú synchronizáciu neskôr alebo má pipeline označiť ako ukončenú chybou. Rozhodovanie prebieha na základe definovaného limitu, ktorý má pipeline na opätovnú synchronizáciu predtým, než bude označená za chybnú a zo synchronizácie vyradená. Informácie o aktuálnom stave sú zo serveru získavané prostredníctvom služby tromi spôsobmi:

- **zo súboru <pipeline_id>.temp** - ktorý obsahuje jobID jednotlivých krokov pipeline
- **prostredníctvom príkazu qacct** - informácie o dokončených krokoch pipeline získavané prostredníctvom ich jobID
- **prostredníctvom príkazu qstat** - informácie o joboch, ktoré ešte neboli dokončené

Okrem aktuálneho stavu sa tiež z výpočtového serveru získava aj súbor s príponou log, ktorý obsahuje aktuálny výpis terminálu, nakoľko je celý výpis každého skriptu presmerovaný práve do tohto súboru. Užívateľ má tak stále podrobný prehľad ako o stave pipeline, tak o aktuálnom výpise.

Po dokončení posledného kroku pipeline sa zaháji príprava dát na prenos výsledkov. To prebieha už spomenutým skomprimovaním zložky res a následným presunutím vytvoreného ZIP archívu do adresára gres, ktorý možno vidieť na obrázku 3.5 a zaháji sa zmazanie pracovného adresára pipeline z disku. Po dokončení záverečných rutín zanechá záznam v súbore s príponou temp, ktorý informuje API o tom, že bolo dokončené vykonávanie pipeline a dáta sú pripravené na prenos.

3.4.6 Prenos výsledných dát

Posledným krokom v životnom cykle pipeline je prenos dát z výpočtového servera na server s API, kde budú výsledné data dostupné na stiahnutie. O prenos sa stará služba PostDataTransferService, ktorá rovnako ako služba DataTransferService beží na pozadí. Jedná sa o rovnaký princíp prenosu dát, ale opačným smerom, teda z výpočtového serveru na server s API. Počas presunu sa vykoná stiahnutie archívu <pipeline_id>.zip z adresára gres a záverečné stiahnutie logu.

V prípade úspešného prenosu sa vykoná skontrolovanie celkového stavu pipeline v databáze. V prípade, kedy nie je možné log alebo výsledné data stiahnuť, napríklad z dôvodu, že žiadne výsledné dáta nemá, nastaví sa pipeline príznak, že k stiahnutiu niektorého zo súborov nedošlo a do logu sa pridá informatívna hláška a súbor ostane v adresári na výpočtovom serveri, nedochádza teda k jeho zmazaniu, aby nedošlo k strate dát, napríklad v prípade chyby spojenia. Dáta následne môže

stiahnuť správca zariadenia manuálne. Po dokončení tohto kroku sa nastaví stav na Dokončená a užívateľovi je prostredníctvom klienta umožnené stiahnuť výsledné dáta.

3.4.7 Neočakávané udalosti

Medzi neočakávané udalosti z pohľadu plánovania a synchronizácie patria pipeline, vykonávané na výpočtovom zariadení, s ktorým bolo prerušené sieťové pripojenie. K takejto udalosti môže dôjsť z viacerých dôvodov. Patrí tu napríklad: zmena konfiguárie zariadenia, či strata sieťového pripojenia, výpadok elektrickej energie a podobne. V takomto prípade vzniká pipeline, o ktorej nie sú žiadne informácie a nie je garantované opätovné spojenie. Aj keď môže tento stav nastať aj v prípade, že joby pipeline bežia korenktne, služba GlobalQueueService aplikuje pesimistický prístup a teda predpokladá, že pipeline skončili chybou. K zmene stavu na chybu však nedochádza hneď. Stav pipeline sa nastaví na Stav neznámy. Služba sa následne pokúša o opätovné pripojenie po určitú dobu, ktorá je nastavená pri konfigurácii API. Po uplynutí tejto doby môže nastať v prípade úspešnej synchronizácie zmena stavu na aktuálny stav. V prípade, že problémy pretrvávajú, pipeline je nastavená hodnota stavu na Chyba a je vyradená zo synchronizácie. Dôvodom tohto riešenia je snaha zbaviť sa zombie procesov, ktoré by zbytočne zaťažovali API, sieť a výpočtové zariadenie, na ktorom prebieha vykonávanie pipeline.

Ďalšou neočakávanou udalosťou v životnom cykle pipeline je neočakávaná chyba. V prípade, že sa v ktoromkoľvek kroku pipeline vyskytne chyba zabraňujúca pokračovaniu v procese vykonávania pipeline, aktuálny stav sa okamžite nastaví na Chyba.

Posledným typom popisovanej neočakávanej udalosti je prerušenie pipeline. Prerušenie je možné vykonať iba zo strany užívateľa alebo administrátora. Hodnota stavu sa v takomto prípade nastaví na Zrušená. Samotné zrušenie pipeline môže prebiehať vo viacerých scenároch. Ako prvé aplikácia počíta s tým, že skript je naplánovaný Grid enginom, preto sa snaží o zastavenie jobov pipeline príkazom qdel.

Pipeline je možné zastaviť iba v prípade, že sa nachádza v niektorom zo stavov medzi stavom Naplánované a stavom Dokončené. Po zrušení, resp. zastavení sa okamžite vykoná zmazanie pracovného adresára pipeline z výpočtového servera, zmazanie výsledných dát a prenos logu. Pipeline je v tomto stave od začiatku vyradená zo synchronizácie, aby nedošlo ku kolízii.

3.4.8 Sledovanie výkonu výpočtového zariadenia, front a hostov

Aby bolo vyťaženie zariadenia čo najnižšie, synchronizácia dát týkajúcich sa výkonu výpočtového zariadenia, hostov a front, rovnako ako aj informácii týkajúcich sa ich aktuálnej konfigurácie Grid enginu prebieha na pozadí prostredníctvom služby GEConfService. Tá môže synchronizovať jedno alebo viacero zariadení paralelne v závislosti na nastavení určenom v konfigurácii API.

Medzi informácie synchronizované na pozadí patria informácie týkajúce sa percentuálneho vyťaženia CPU, RAM, celkového množstva voľného priestoru na disku, ako aj informácii týkajúcich

sa konfigurácie daného hosta a dostupných zdrojov, akými sú napríklad: počet procesorov, jadier procesorov, vláken, atď.

Aj napriek tomu musí byť časť informácii získavaná mimo Grid enginu, časť informácii môže byť získavaná iba priamym pripojením k zariadeniu, nakoľko je pre ne potrebný prístup k rozhraniu terminálu konkrétneho zariadenia - konkrétnejšie sa jedná o informácie týkajúce sa percentuálneho vyťaženia každého z procesorov ako samostatnej jednotky, ktoré sú získavané zo súboru /proc/cpuinfo a sledovanie voľného priestoru vrámci jednotlivých partícii, na ktoré je disk rozdelený, získavané využitím príkazu df. Hlavný problém v tomto prípade predstavujú host zariadenia, ku ktorým je povolený iba obmedzený prístup prostredníctvom Grid enginu. Správca systému preto môže vrámci systému pridať prístupové údaje ku každému z hostov samostatne, čím odomkne možnosť detailného sledovania výkonu host zariadenia. Pričom treba brať do úvahy, že takéto sledovanie zariadenia môže vyvolať jeho čiastočné spomalenie, ktoré sa odvíja do počtu užívateľov sledujúcich detaily zariadenia. Z testov vyplynulo, že sa rádovo jedná o záťaž približne desatinu 0.10% výkonu 1 CPU na 1 užívateľa v prípade silnejších CPU.

3.4.9 Ostatné služby API

Okrem už spomínaných služieb, sa vyskytuje ešte jedna služba, tá zabezpečuje odstraňovanie súborov zo serveru. Mazanie súborov prebieha vrámci databázy pridelením príznaku zmazaný (hodnota parametru del sa nastaví na true). K fyzickému zmazaniu súboru z disku dochádza až po uplynutí doby, ktorá je nastavená službe GarbageCollectionService bežiacej na pozadí. Predvolene bola zvolená hodnota 31 dní. Užívateľ môže po pridelení príznaku zmazaný súbor kedykoľvek počas doby 31 dní nájsť skript alebo pipeline v sekcii Kôš a obnoviť ho.

Aby bolo zaistené, že nedôjde ku kolízii pri fyzickom zmazaní a pokuse o obnovu súboru, služba starajúca sa o fyzické mazanie má oproti tomuto limitu posun o 1 deň. Pričom dĺžku tejto doby je možné nastaviť počas konfigurácie pred nasadením aplikácie na server. Dochádza teda k približne jednodňovému rozdielu medzi virtuálnym a fyzickým zmazaním súboru. Pričom pod virtuálnym zmazaním možno rozumieť zmazanie, kedy systém súbor zobrazuje ako už zmazaný i keď je ešte fyzicky prítomný na disku. Po fyzickom zmazaní súboru zo serveru sa nastaví položke príznak zmazaný zo serveru (hodnota parametru sdel sa nastaví na true). To, kedy sa má súbor fyzicky , sa určuje na základe času a dátumu poslednej zmeny uloženého v databáze, nakoľko súbor, ktorý je určený na zmazanie, už nie je možné upravovať.

3.4.10 Odolnosť voči útoku

Okrem zraniteľnosti v podobe prítomnosti prihlasovacích údajov a adresy servera, na ktorom beží Grid engine, možno hovoriť aj o zraniteľnosti, respektíve odolnosti v prípade DDOS útoku. Zatiaľ, čo v prípade útoku na aplikáciu bežiacu v lokálnom režime by došlo k ovplyvneniu HPC s vysokým výkonom, v prípade zariadenia bežiaceho v režime vzdialeného ovládania by spomalenie malo vplyv

len na server, na ktorom beží API s klientom, eventuálne klienta samotného. Klienta a API je totiž možné rozdeliť, umiestniť na rozdielne servery a následne prepojiť prostredníctvom nastavenia proxy modulu vo webpacku klientskej aplikácie.

Okrem zraniteľnosti voči DDOS útokom by sa dalo hovoriť aj o zraniteľnosti z pohľadu kódu, ktorý je spúšťaný vrámci jobov (tvoriacich pipeline) prostredníctvom Grid enginu. Môže totiž dôjsť napríklad k tomu, že užívateľ naplánuje skript, ktorý vyťažuje hardware zariadenia na maximum, či dokonca v horšom prípade poškodzuje hardware zariadenia. Medzi takéto skripty patrí napríklad ťažba virtuálnej meny (bitcoin, litecoin a podobne) alebo kód, respektíve súbor kódov, ktorých jediným účelom je jednoducho spotrebovať celý výkon, respektíve zdroje zariadenia. Takým príkladom je v lepšom prípade kód obsahujúci nekonečný cyklus, ktorý spotrebuje všetky dostupné zdroje zariadenia - pamäť, výkon CPU a podobne. V horšom prípade sa môže jednať o kód, ktorý svojou činnosťou spôsobí materiálne škody, napríklad zničenie SSD diskov neustálym zápisom či prepisom ich obsahu.

Ďalšie riziko predstavuje možnosť, že užívateľ nenahrá na server vírus samotný, ale ako jeden z jobov naplánuje skript, ktorý vykoná stiahnutie a následné spustenie vírusu.

Niektoré z týchto prípadov by bolo možné detekovať a eliminovať využitím metód na detekciu malware, neurálnymi sieťami, analýzou kódu či vyťaženia zariadenia. Mnohé z týchto možností by boli veľmi ťažko realizovateľné. U iných ako napríklad u analýzy vyťaženia zariadenia by bolo veľmi obtiažne určenie hranice, kedy sa jedná o škodlivý kód a kedy nie. Riešením by mohla byť kombinácia viacerých zo spomínaných techník. Poslednou možnosťou, ktorá bola zároveň zvolenou cestou pri vývoji aplikácie je dôvera užívateľovi, že na server takéto skripty nenahrá, respektíve nebude takéto skripty spúšťať. Ďalej využitie obmedzení zdrojov vrámci front, prípadne rozdelenie zariadenia na viacero hostov. Poslednou možnosťou je obmedzenie prístupu k určitým frontám užívateľom v závislosti na dôveryhodnosti.

Okrem dôvery užívateľovi možno eliminovať možnosť niektorého zo spomenutých scenárov napríklad správnymi nastaveniami na strane výpočtového zariadenia. Napríklad v prípade spojenia cez vzdialený prístup (SSH + SFTP alebo SSH + sieťový adresár) využiť užívateľské konto, ktoré nemá administrátorské práva vrámci systému a vrámci Grid enginu má práva užívateľa, ktorému je umožnené iba spúšťať a zastavovať vlastné joby. Či použiť režim API, v ktorom sa od každého z užívateľov vyžaduje prístup cez vlastné užívateľské konto.

3.4.11 Vplyv vonkajších faktorov na vykonávanie pipeline

Ďalším z problémov, ktorý bolo potrebné vyriešiť v spojení vykonávaním pipeline sú vonkajšie vplyvy, pod ktorými rozumieme vplyv uživateľom naplánovaných skriptov na joby, z ktorých sa pipeline skladá, prípadne na pipeline iných užívateľov. V tomto prípade je miera, s akou môže užívateľ zasahovať do konania pipeline závislá na právach, ktoré boli vrámci zariadenia pridelené užívateľovi v dobe spúšťania skriptov.

V prípade, že vrámci konfigurácie zariadenia v prostredí aplikácie nastaví administrátor príznak požadovať prihlásenie s vlastným kontom, bude užívateľov vplyv na joby ostatných užívateľov závislý na právach, ktoré sú im pridelené vrámci Grid engine daného zariadenia. Keďže pod vonkajším vplyvom možno rozumieť tiež fyzické zmazanie súborov, či už skriptov tvoriacich pipeline alebo dát, s ktorými môžu skripty pipeline pracovať. Obmedzenie prístupu v tomto smere rieší utilita setAcl, ktorá umožní užívateľovi vykonávať mazanie a úpravy iba so súbormi, ktoré sú vytvorené jeho účtom, ktorý má vrámci zariadenia vytvorený. Výnimku v tomto smere predstavuje účet, prostredníctvom ktorého je vykonávaná synchronizácia. U neho sa totiž predpokladá, že bude mať vrámci pracovného adresára aplikácie nastavené vyššie užívateľské práva, nakoľko cez toto konto prebieha synchronizácia, ale aj zastavovanie pipeline, jobov, následné zmazanie nepotrebných súborov a adresárov umiestnených vrámci adresára aplikácie.

3.4.12 Obmedzenie prístupu k frontám a hostom

Posledným z rozoberaných bezpečnostných aspektov aplikácie je prístup k frontám a hostom. Keďže vrámci Grid engine nie je možné obmedziť prístup užívateľom k hostom, ale iba k frontám. Nie je možné nijak kontrolovať, na ktorom z execution hostov sa bude job vykonávať v prípade, že výber fronty a hosta užívateľ neuskutoční. V takomto prípade totiž kontrolu nad výberom preberá Grid engine.

V rámci riešenia bolo rozhodnuté, aby obmedzenie prebehlo na strane Grid engine v prípade, že od užívateľa bude požadované prihlásenie prostredníctvom vlastného účtu. V prípade, že administrátor v takomto prípade nechce umožniť užívateľovi prístup k niektorému z hostov, neumožní mu prístup k frontám, ktoré sa na danom hostovi nachádzajú.

Pravidlá prístupu k jednotlivým frontám a hostom sa do systému premietnu pri synchronizácii, počas ktorej prebieha aj synchronizácia týchto nastavení. V prípade, že by chcel administrátor systému predsa len skryť frontu pre daného užívateľa vrámci systému, môže tak učiniť v sekcii Fronty. Problémom tohto riešenia je však to, že užívateľovi síce nebude umožnené vidieť frontu vrámci systému, nič však nezabráni Grid enginu, aby si vybral túto frontu v prípade, že výber fronty ponechá užívateľ na Grid engine.

3.5 Generovanie formulárov

Keďže každý zo skriptov môže vyžadovať iné vstupné parametre, ktoré sa od seba navzájom líšia názvami alebo dátovými typmi a pri každom spustení sa očakáva iná kombinácia hodnôt vstupných parametrov, bolo potrebné nájsť spôsob ako efektívne vytvárať a následne opakovane generovať formuláre umožňujúce užívateľovi zadávať potrebné vstupné hodnoty. S týmto cieľom bol vytvorený balíček s názvom json-obj-form-generator, ktorý je robený priamo na mieru tak, aby spĺňal

požiadavky a potreby vytváranej aplikácie. Balíček sa skladá z dvoch časti: generátora formulárov a dizajnéra.

3.5.1 Generátor formulárov

Jedná sa o časť balíčka, ktorá zabezpečuje generovanie formulárov a validáciu vstupov. Aktuálne sú implementované a podporované v plnom rozsahu, teda vrátane validácie, vstupy numerické (float a integer), textové, boolean, ale aj vstupy užívateľom definovaného typu. Pričom v prípade užívateľom definovaných typov je možné typy definovať dvomi spôsobmi:

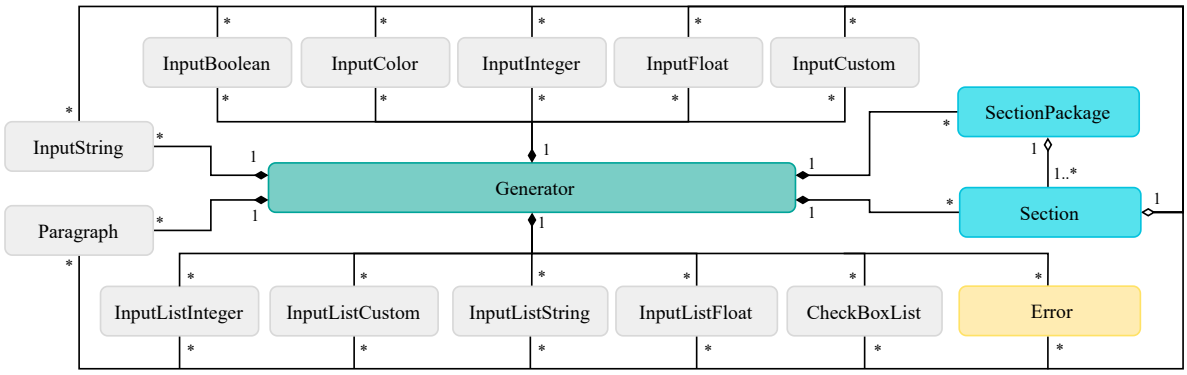
- **Vymenovaním hodnôt**

  Prebieha vymenovaním všetkých hodnôt daného dátového typu, ktoré môže vstup nadobúdať. Z týchto hodnôt sa následne vygeneruje list, z ktorého si na základe nastavení môže užívateľ zvoliť jednu, prípadne viaceré hodnoty.

- **Validačným pravidlom**

  Definícia dátového typu pomocou validačného pravidla sa líši od vymenovania hodnôt tým, že sa namiesto vymenovania hodnôt zadá pravidlo, podľa ktorého sa má vstupná hodnota validovať. Samotná validácia následne prebieha prostredníctvom funkcie regex.

Ako bolo už spomenuté a ako tiež môžno vidieť na obrázku 3.13, balíček podporuje vstupy jedno i viac hodnotové. Umožňuje tiež export aktuálnych nastavení do formátu JSON, ktorý je spätne možno použiť na opätovné vygenerovanie formulára aj s už zadanými hodnotami. Na tvorbu samotných formulárov však primárne slúži plnohodnotný dizajnér, ktorý je súčasťou knižnice.



Obr. 3.12: Schéma fungovania generatora knižnice json-obj-form-generator

Princíp fungovania generovania formulárov môžno vidieť na obrázku 3.12 z ktorého vyplýva, že generovanie začína v komponente Generator. Prebieha výberom prototypov objektov z poľa, pričom následný prenos dát medzi jednotlivými úrovňami prebieha prostredníctvom spätného volania metódou onChange. Medzi hlavné výhody pritom patrí možnosť ľahkého a rýchleho rozšírenia o nové

funkcie podľa vlastných potrieb, ako aj to, že obsahuje iba prvky, ktoré aplikácia reálne využije. Neobsahuje teda prvky, ktoré by aplikácii zbytočne pridávali na veľkosti či ju inak spomaľovali. Medzi vypustené prvky patrí napríklad generovanie kalendárov, máp a podobne. Niektoré z nich však môžu byť časom pridané, nakoľko sa na knižnici neustále pracuje. Objekty tvoriace formulár možno rozdeliť na tri typy:

- objekty generujúce vstupné polia, do ktorých môže užívateľ zadávať svoje vstupy
- list sekcii (SectionPackage) spájajúci sekcie na vizuálnej úrovni
- sekcia, ktorá nie je síce vizuálne rozlíšená od zvyšku formuláru, môže mať však vplyv na výsledný objekt za istých nastavení

Rozdiel medzi listom sekcii a klasickou sekciou je skôr v dizajne a komponentách použitých pri renderovaní.

Obr. 3.13: Ukážka generovania formulárov pomocou balíčka json-obj-form-generator

O samotné generovanie jednotlivých vstupných polí sa starajú komponenty Generator a Section, ktoré na základe užívateľom definovaného typu generujú, respektíve spájajú rôzne typy vstupných polí spolu s popiskom, názvom a chybovými hláškami.

O validáciu sa následne stará v prevažnej časti obvykle komponenta, ktorá zabezpečuje rendrovanie daného typu vstupu. Validácia pritom prebieha vrámci metódy onChange a metódy getDerivedStateFromProps. Metóda onChange zároveň po validácii pošle data rekurzívne o úroveň vyššie až do samotnej hlavnej komponenty - Generator, odkiaľ ich môže užívateľ získať použitím metódy onChange. Väčšina komponentov, ktoré obsahuje balíček, majú iba metódu render a metódu onChange zabezpečujúcu prenos dát na vyššiu vrstvu. Všetky dáta formuláru sú urdžiavané v koreňovej komponente Generator, odkiaľ sú zmeny distribuované do listov stromovej štruktúry - jednotlivým poliam vstupov.

Okrem jednotlivých komponent validuje vstupy tiež komponenta Generator, ktorá pred prvým vygenerovaním formuláru, teda počas inicializácie objektu, skontroluje vstupný objekt JSON a vráti stav jeho validácie a aktuálnu hodnotu.

Výhodou oproti ostatným knižniciam je možnosť vrátiť buď lineárny objekt, stromovo štruktúrovaný lineárny objekt alebo stromovú štuktúru objektov. Príklad použitia knižnice môžno vidieť na obrázku 3.13. Balíček bol publikovaný na NPM, kde je voľne dostupný na adrese `https://www.npmjs.com/package/json-obj-form-generator` na stiahnutie.

3.5.2 Dizajnér

Hoci je generátor na dizajnérovi úplne nezávislý, v opačnom smere tento vzťah úplne neplatí. Medzi komponentami dizajnéru a generátoru sú dva typy závislostí. Jednou a zároveň najväčšou závislosťou je to, že dizajnér generuje objekty typu JSON, ktoré musí byť generátor schopný vygenerovať. Druhý predstavuje samotná konštrukcia dizajnéra, ktorý využíva generátor na vizualizáciu zatiaľ vytvoreného formulára, prípadne jeho podčastí, aby mal užívateľ tvoriaci formulár predstavu o tom, ako bude fungovať, ako sa bude správať a vyzerať. Príklad môžno vidieť na obrázku 3.15, kde môžno vidieť príklad dizajnéra.

V pravom hornom rohu možno vidieť dvojicu tlačidiel, zelené tlačidlo umožňuje zobraziť celý formulár tak, ako by ho vygeneroval komponent Generator. Modré slúži na zobrazenie objektu JSON v textovej podobe v dialogovom okne, kde ho možno skopírovať jednoduchým kliknutím do schránky. Tlačidlami pridať komponent je možné pridať na dané miesto formulára nový komponent, ktorý je možno vybrať zo zoznamu. Každá vložená komponenta má v pravej časti umiestnený svoj typ s príslušnou ikonou, pod ktorým nasledujú jej parametre, ktorými užívateľ definuje správanie daného vstupného poľa alebo časti formulára.

V ľavej časti potom vidieť štvoricu tlačidiel. Dve z nich slúžia na posúvanie komponenty v poradí nahor a nadol. Tretie tlačidlo (vyznačené oranžovým štvorčekom) slúži na zobrazenie časti formulára tak, ako by vyzeral po vygenerovaní generátorom. Príklad aktivovanej funkcie možno vidieť vyznačený ružovou farbou. Posledné tlačidlo slúži na vymazanie komponenty z formulára v prípade sekcie a balíčku sekcii spolu s objektmi, ktoré obsahuje.

Medzi najdôležitejšie parametre sprístupňujúce funkcionality patria:

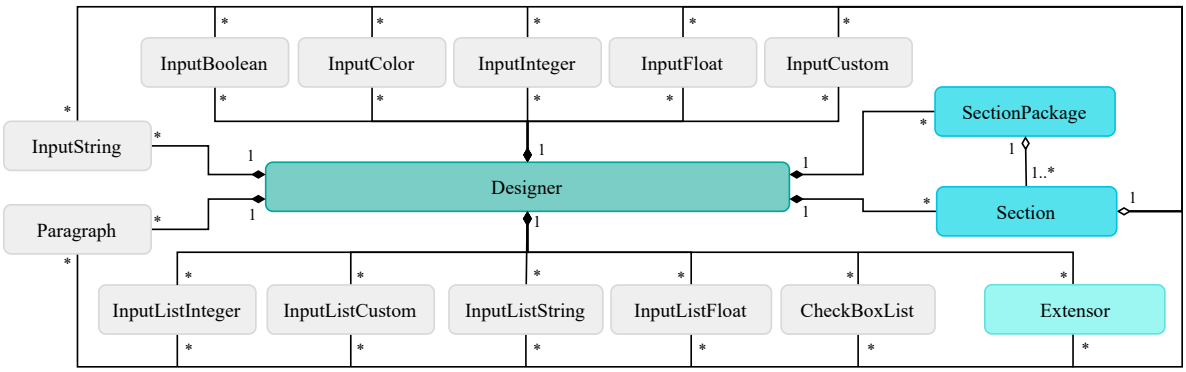
- **export** - povolí zobrazeniť výsledný objekt JSON, ktorý slúži na generovanie formulára v textovej podobe v dialógovom okne, kde ho možno skopírovať do schránky (funkcie poskytované modrým tlačidlom umiestneným vo vrchnej časti dizajnéra)
- **extended** - sprístupní možnosť zobraziť prvky tak, ako budú vygenerované generátorom, prípadne zobraziť rovno celý formulár priamo z dizajnéru (funkcie poskytované zeleným tlačidlom umiestneným vo vrchnej časti dizajnéra a tlačidlom vyznačeným oranžovou farbou)
- **json** - JSON objekt generujúci formulár (môže byť zadaný v prípade, že chce užívateľ upravovať už existujúci formulár)
- **mode** - typ serializácie objektu obsahujúceho hodnoty navrátené generátorom
- **title** - názov dizajnéra
- **translation** - umožňuje preložiť dizajnér do vlastného jazyka vrátane chybových hlásení

Medzi vlastnosti, ktoré možno parametru definovať, patrí napríklad to, či je daný parameter povinný alebo nepovinný, ako aj možnosť nastavenia predvolenej a prednastavenej hodnoty, minimálnej a maximálnej hodnoty (v prípade textového vstupu minimálnej a maximálnej dĺžky). Rozdiel medzi predvolenou a prednastavenou hodnotou je pritom ten, že prednastavená hodnota je doplnená do vstupného poľa počas vygenerovania. Zatiaľ čo predvolená hodnota bude nastavená premennej v prípade, že je vstupné pole nevyplnené a označené ako povinné. V takom prípade sa však hodnota v generátore nezobrazí vo vstupnom poli, ale užívateľ je upozornený, že dôjde k aplikovaniu predvolenej hodnoty poznámkou pod vstupným poľom - tak ako to môžeme vidieť na obrázku 3.15 vyznačené červenou farbou.

Oba typy dizajnéru, rovnako ako generátor, vykonávajú pritom validáciu predvolenej a prednastavenej hodnoty, aby bolo zaručené, že bude formulár vygenerovaný s validnými hodnotami a bez chýb. Validáciu v tomto smere predstavuje metóda isValid, ktorá pri každej zmene ktoréhokoľvek poľa dizajnéru vráti stav validácie objektu JSON, ktorý je vrátený metódou onChange.

Vrámci formulára možno vstupné polia deliť do sekcii a podsekcii. Podsekcie pritom môžu byť tvorené rekurzívne prakticky bez obmedzenia. Jediné obmedzenie v tomto smere čiastočne predstavuje CSS štýlovanie, ktoré je preddefinované priamo v balíčku iba do 2. úrovne rekurzie.

Okrem primitívnych typov a typov definovaných užívateľom, ktoré sú definované vymenovaním hodnôt či definovaním regexu, umožňuje knižnica definovať viachodnotové vstupy, ktoré sú realizované formou checkbox listov alebo listov so vstupnými poliami, prípadne rozbaľovacími zoznamami. Poslednými typmi sú typy radené do kategórie ostatné. Jedná sa o dátumy, farby, čas a podobne, pričom im obvykle možno definovať iba tzv. placeholder - čiže popisok vstupného poľa, informačný popisok, názov, typ, prednastavenú a preddefinovanú hodnotu, v niektorých prípadoch tiež minimálnu a maximálnu hodnotu.



Obr. 3.14: Schéma fungovania dizajnéra knižnice json-obj-form-generator

Schému fungovania samotného dizajnéru možno vidieť na obrázku 3.14. Pri hlbšej analýze jednotlivých tried dizajnéra si možno povšimnúť, že všetky komponenty majú mimo parametrov súvisiacich s daným dátovým typom takmer totožné vstupné parametre. Každá z komponent má zadefinovanú metódu isValid a onChange, ktoré sa starajú o vnútornú komunikáciu hlavnej komponenty s podkomponentami a validáciu navrátených objektov. Jedným z výrazných plusov je fakt, že dizajnér plne podporuje preklad jednotlivých polí. Je teda možné vytvoriť si dizajnér s vlastnými popismi a názvami polí, resp. preložiť si dizajnér do vlastného jazyka. V prípade, že tak užívateľ neučiní, je defaultne dizajnér generovaný v anglickom jazyku.

Preklad dizajnéra je pritom možné jednoducho vygenerovať prostredníctvom generátora prekladov aj na stránke projektu dostupnej na adrese:
`https://radovan-pranda.github.io/json-obj-form-generator`.

Okrem vracania JSON objektov, ktoré je možné generovať pomocou komponenty Generator, podporuje dizajnér tiež spätné generovanie, čiže z JSON objektu je možné vygenerovať dizajnér, pomocou ktorého možno tento objekt znovu upravovať.

3.5.3 Porovnanie s ostatnými knižnicami

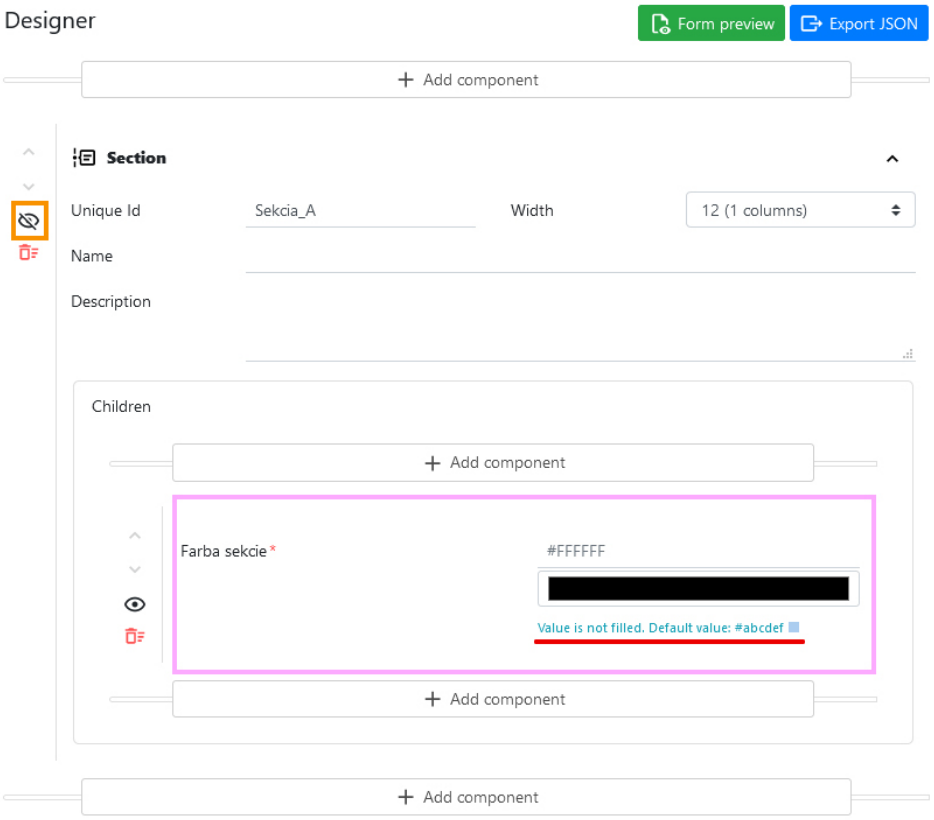
Vrámci kapitoly 2.3 boli rozoberané aspekty, súčasti a funkcie poskytované existujúcimi knižnicami, ktoré sú schopné generovať formuláre z JSON objektov. Prieskum a testovanie bolo zavŕšené porovnaním, ktoré ako možno vidieť v podkapitole 2.3.4 ukázalo, že sa knižnice nachádzajú približne na rovnakej úrovni.

Na základe poznatkov z testovania a potrieb vytváranej aplikácie vznikala vyššie popisovaná knižnica. Ako bolo možno sa dočítať v prieskume žiadna z testovaných knižníc neposkytovala dizajnér, ktorý by bol priamo súčasťou knižnice. Každá z nich však poskytovala spomínaný dizajnér, ktorý by zjednodušil vytváranie formulárov implementovaný vrámci dokumentácie.

Výhodou niektorých knižníc oproti vytváranej je podpora tém, nakoľko vyvinutá knižnica poskytuje iba malú časť štýlovania, ktoré skôr upravuje rozloženie jednotlivých častí. Štýlovanie jednotli-

vých komponent zabezpečuje knižnica reactstrap, ktorej časť prvkov bola použitá vrámci vytváranej knižnice. Tento fakt však nielenže nepredstavuje vážny nedostatok, respektíve nevýhodu, ale umožňuje každému skúsenejšiemu užívateľovi vytvoriť vlastné štýlovanie s využitím front-end frameworku Bootstrap, prípadne použiť niektorú z Bootstrap tém.

Naopak výhodou oproti ostatným balíčkom je možnosť využitia verzie dizajnéra, ktorý nie je poskytovaný ani jednou z testovaných knižníc. Rovnako ako možnosť jeho prispôsobenia v podobe prekladu do vlastného jazyka. Poskytuje tak ľuďom, ktorí sa ju rozhodnú implementovať a použiť ju vo svojej aplikácii, poskytovať používateľom svojej aplikácie, možnosť vytvárať JSON objekty umožňujúce generovať formuláre priamo za behu aplikácie.



Obr. 3.15: Ukážka dizajnéra formulárov z balíčka json-obj-form-generator

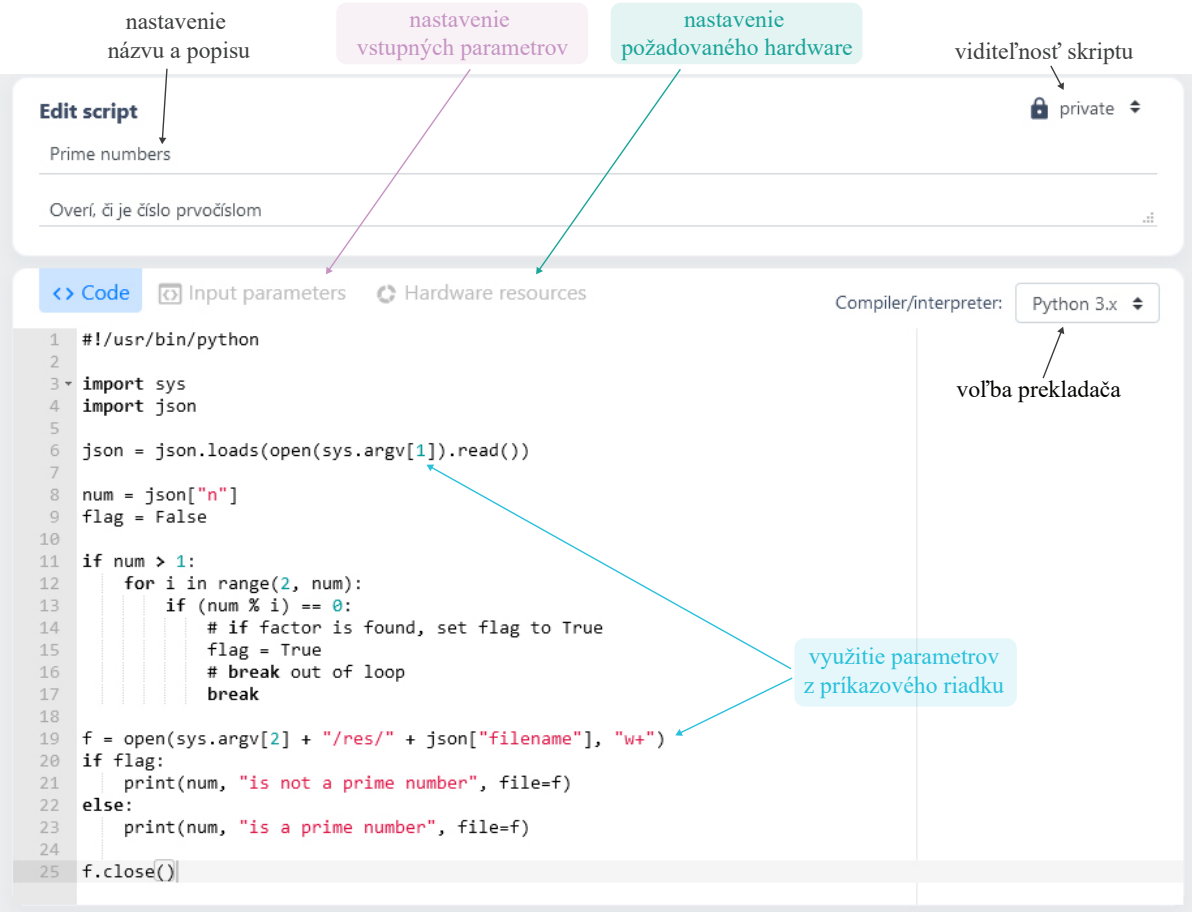
3.6 Klient

Keďže API neposkytuje užívateľsky prívetivé prostredie, bolo potrebné vytvoriť UI umožňujúce pohodlnú komunikáciu a prácu s API poskytovanými funkciami.

Klient získava od API informácie pomocou metódy XMLHttpRequest, pričom dáta vracané API sú serializované do formátu JSON. Vzhľadom na to, že API a klient bežia na rozdielných portoch, spojenie medzi aplikáciami prebieha vďaka nastaveniu proxy.

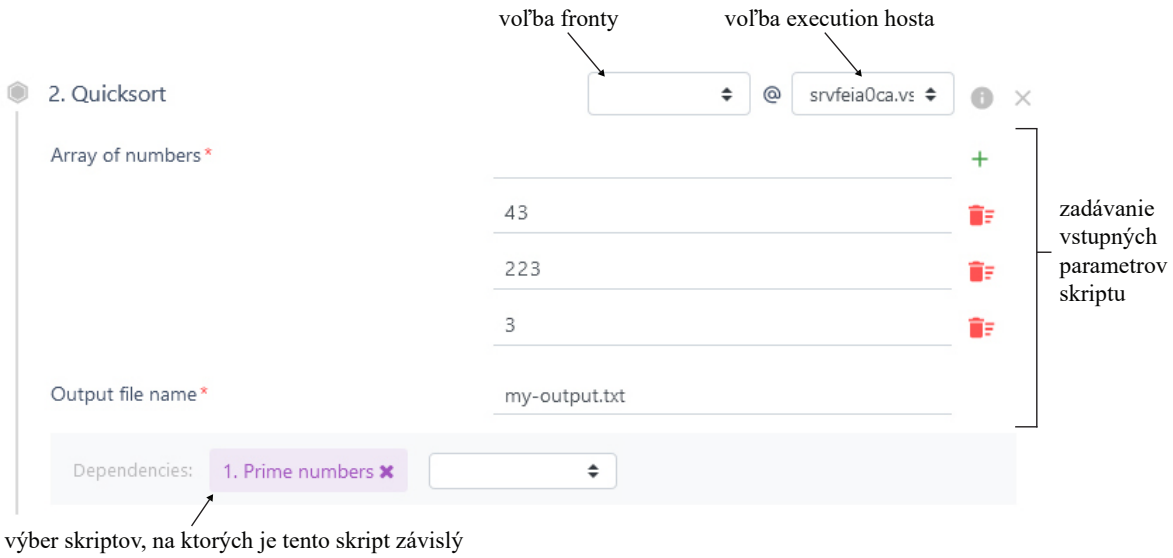
Ako už bolo spomenuté v podkapitole 3.3, na vývoj bola použitá knižnica React. Pre jeho potreby bola pritom vytvorená knižnica umožňujúca generovanie formulárov z objektov typu JSON, o ktorej vývoji pojednáva podkapitola 3.5. Okrem týchto knižníc boli pri vývoji tiež použité knižnice: CoreUI a ReactStrap, ktoré slúžili na jednoduché vytvorenie pekného vizuálu aplikácie. Keďže je knižnica CoreUI platená, nebola použitá jej PRO verzia, ale osekaná základná verzia.

3.6.1 Manažment pipeline template a skriptov



Obr. 3.16: Ukážka editácie skriptov

Jednou z mnohých funkcii poskytovaných aplikáciou je možnosť nahrávania a editácie skriptov, z ktorých je možné následne vytvárať pipeline template. Aplikácia preto obsahuje editor kódu, ktorý je realizovaný prostredníctvom knižnice React Ace, ktorá z časti podporuje farebné zvýraznenie syntaxe na základe zadaného modu. Pod modom sa v tomto prípade rozumie názov jazyka, v ktorom je kód napísaný, čo robí kód z časti prehľadnejším. Ako tiež vidieť na obrázku 3.16, prostredie okrem editora kódu obsahuje aj dizajnér formulára, prostredníctvom ktorého možno zadať vstupné para-



Obr. 3.17: Definovanie kroku pipeline

metre daného skriptu a definíciu zdrojov, ktoré budú požadované od Grid engine pri naplánovaní. Obe tieto sekcie sú realizované vďaka knižnici popisovanej v kapitole 3.5.

Vytváranie pipeline template prebieha rovnako ako v prípade plánovania novej pipeline, ktoré bude popísané v kapitole 3.6.2. Rozdielom je v tomto prípade, že u template užívateľ nevolí frontu, hostov ani zariadenie.

3.6.2 Naplánovanie novej pipeline

Prvým krokom v procese plánovania pipeline predstavuje prostredie klienta. Užívateľ vrámci sekcie Pipelines - New pipeline môže zadefinovať parametre novej pipeline. Patrí medzi ne názov, ktorý nemá nijakú súvislosť s plánovaním, slúži iba užívateľovi na jednoduchšiu identifikáciu počas sledovania behu pipeline vrámci systému.

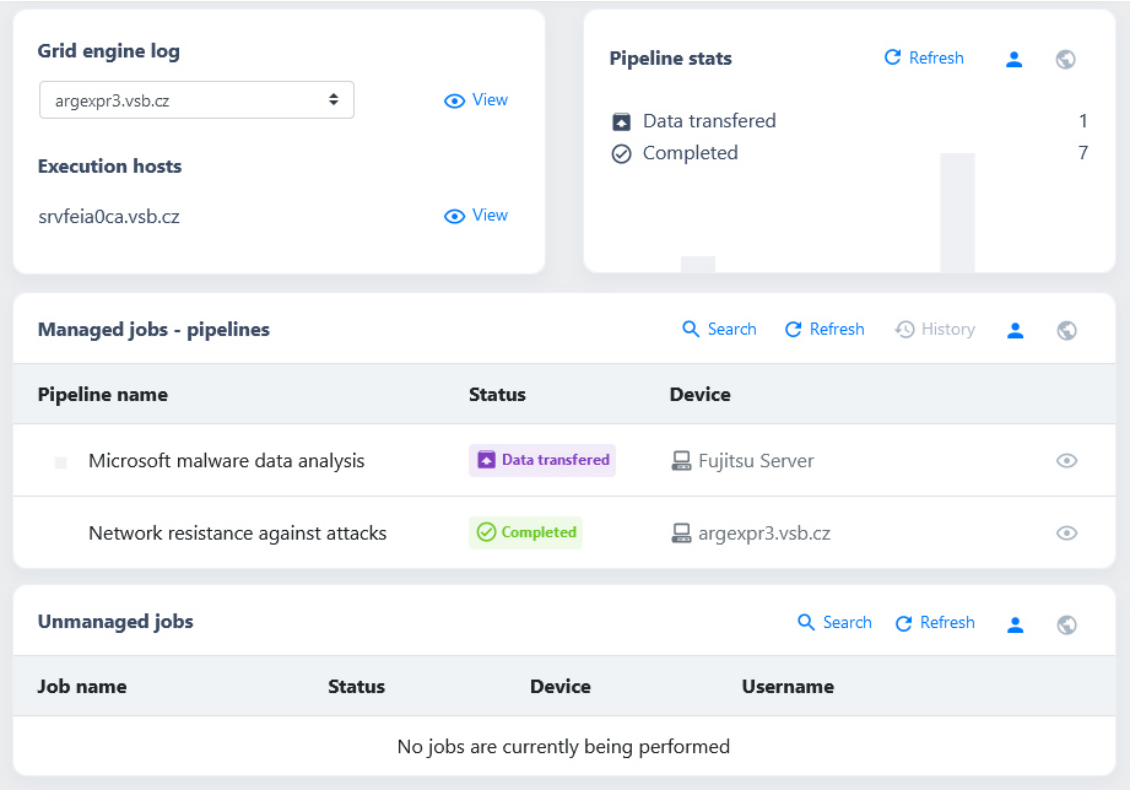
Ďalej musí užívateľ pridať na časovú os skripty, ktoré budú naplánované. Medzi skriptami môžu byť závislosti, ako bolo popísané v kapitole 3.4.4. Vzťahom medzi skriptami môžeme rozumieť, závislosť nasledujúceho skriptu na výstupe predchádzajúceho - napr. z dôvodu predspracovania dát. Definovaním závislosti dáva užívateľ API vedieť, že má vrámci Grid enginu job, vrámci ktorého prebieha vykonávanie skriptu so závislosťou, čakať na dokončenie skriptu, na ktorom je závislý, viď obrázok 3.17. Vykonáva sa tak prostredníctvom nastavenia parametru hold_jid pri naplánovaní nového jobu.

Skripty môžu byť pridávané po jednom alebo nahraním pipeline template, ktorý si užívateľ pripravil vopred. Následne je užívateľ vyzvaný, aby zadal hodnoty vstupných parametrov, ktoré boli označené ako povinné užívateľom, ktorý skript vytvoril. Užívateľovi sa zobrazia minimálne požiadavky kladené na zdroje z pohľadu pipeline.

Na záver si užívateľ musí zvoliť zariadenie, na ktorom bude pipeline naplánovaná. V prípade, že si zvolí vo vrchnej časti stránky predvolenú frontu alebo hosta, bude pre každý zo skriptov defaultne zvolená práve tá, ktorú vybral užívateľ vo vrchnej časti stránky. Predvolené nastavenie nebude použité v prípade, že u daného skriptu užívateľ zvolí inú frontu alebo hosta. V prípade, že užívateľ nezvolí ani v prípade predvoleného nastavenia, ani v prípade nastavenia konkrétneho skriptu žiadnu frontu alebo hosta, o výber sa postará Grid engine.

3.6.3 Monitorovanie jobov a pipeline

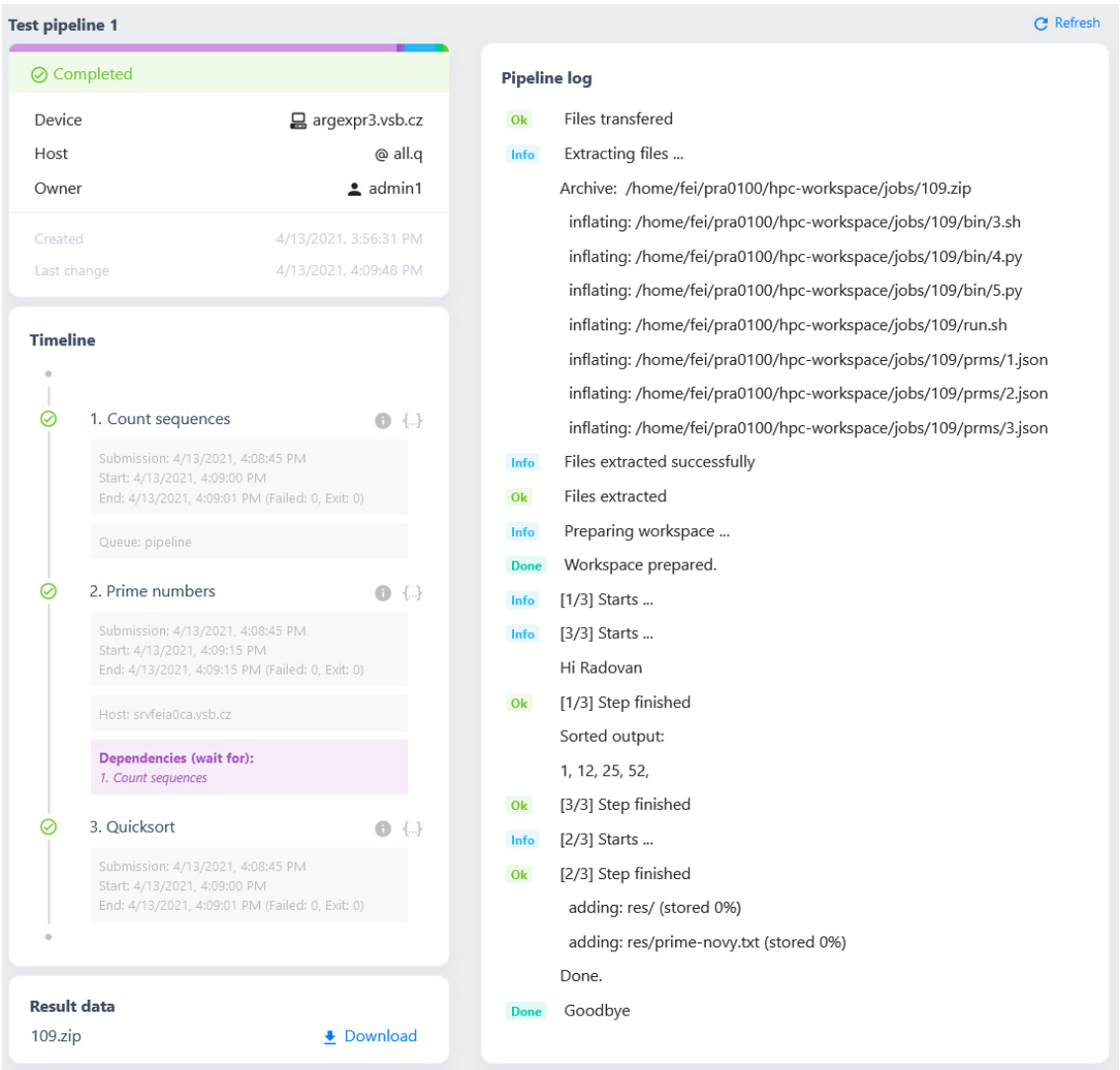
Po naplánovaní pipeline môže užívateľ sledovať ich stav a priebeh vrámci hlavného panela, viď obrázok 3.18. Vo vrchnej časti stránky sú umiestnené logy master a execution hostov, u ktorých je daný užívateľ v roli manažéra alebo operátora. Okrem toho je vrchná časť tiež vyhradená krátkej štatistike týkajúcej sa stavov, v ktorých sa nachádzajú pipeline naplánované za posledný mesiac.



Obr. 3.18: Ukážka hlavného panelu

V spodnej časti stránky sa nachádza zoznam pipeline, ktoré zatiaľ neboli dokončené, prípadne boli dokončené nedávno. Sú tu užívateľovi poskytnuté základné informácie týkajúce sa pipeline, ale aj nemanažovaných jobov. Joby sú tu rozdelené do dvoch kategórii, ako už bolo uvádzané v kapitole 3.4. Okrem informačnej hodnoty tiež možno v tejto časti panelu pipeline alebo job zastaviť, či zobraziť ich detaily.

Vrámci detailu pipeline možno sledovať nielen detaily týkajúce sa spúšťanej pipeline, ale v prípade, že pipeline ešte stále nebola dokončená aj detaily týkajúce sa hostov, front a vyťaženia zariadenia, na ktorom bola pipeline naplánovaná. Po dokončení vykonávania a prenosu dát na stranu API je užívateľovi umožnené stiahnutie výstupných dát. Ako vidiet na obrázku 3.19, možno tiež v tejto sekcii aplikácii nájsť aj vstupné data zadané užívateľom a štruktúru pipeline, teda postupnosť, v ktorej majú byť naplánované skripty a vzťahy medzi nimi.

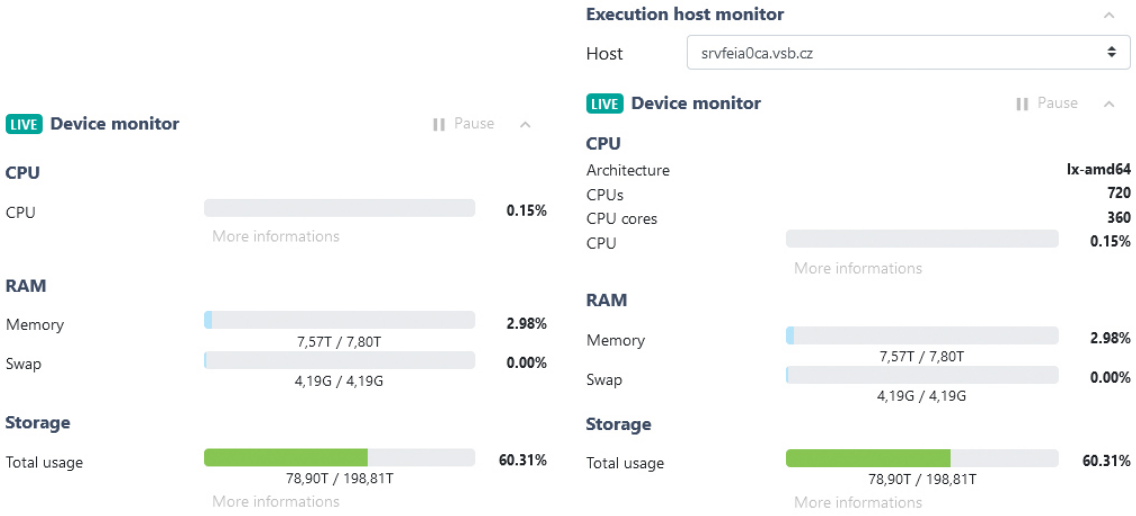


Obr. 3.19: Ukážka detailu pipeline

3.6.4 Monitorovanie výkonu a zdrojov zariadenia

Okrem vytvárania a sledovania pipeline je možné sledovať prostredníctvom klienta aj aktuálny výkon a spotrebu zdrojov master a execution hostov. Konkrétne sa jedná o informácie týkajúce sa spotreby disku, pamäte a výkonu CPU v percentách.

Synchronizácia základných informácii prebieha vrámci API na pozadí. V reálnom čase prebieha komunikácia so zariadením iba v prípade, že chce užívateľ sledovať aktuálne vyťaženie každého CPU ako jednotky alebo každej z particii disku. Vzhľadom na to, že informácií poskytovaných Grid enginom je viac, sú informácie týkajúce sa execution hostov podrobnejšie, ako vidieť aj na obrázku 3.20.



(a) master host (b) execution host

Obr. 3.20: Ukážka monitorovania zdrojov zariadení

Kapitola 4

Testovanie

Poslednou kapitolou pred samotným záverom je testovanie výslednej aplikácie, ktorá sa venuje problémom spojenými s konfiguráciou, inštaláciou Grid enginu - teda prípravami na testovanie a samotnému testovaniu výslednej aplikácie.

	HPC VŠB	**VM s centOS**	**Fujitsu Server**
CPU	Intel(R) Xeon(R) Gold 6154 3.7GHz (720CPUs)	Intel(R) Core i7-8565U 1.8GHz (4CPUs)	Intel(R) Xeon(R) E3-1225 3.3GHz (4CPUs)
Pamäť	RAM: 7.3TB, Disk: 7734TB	RAM: 2GB, Disk: 40GB	RAM: 8GB, Disk: 8TB
OS	Red Hat EL7 64bit	CentOS 7 64bit	Windows Server 2019 64-bit
Ostatné	SGE 8.1.7 (SoGE)	SGE 8.1.7 (SoGE)	API

Tabuľka 4.1: Špecifikácia zariadení, na ktorých bolo vykonávané testovanie

4.1 Problémy spojené Grid enginom

Prvým zásadným problémom, ktorý nastal ešte pred samotným vývojom bola problémová inštalácia. I keď je SoGE relatívne funkčný systém, má svoje nedostatky hlavne v oblasti dokumentácie a inštalácie.

Problém s inštaláciou

Hlavným problémom SoGE spájajúcim sa inštaláciou je, že inštalačné skripty nie sú dokonale odladené a je potreba ich pred inštaláciou upraviť do funkčnej podoby. Najčastejším dôvodom úprav boli pritom problémy s právami, kedy operácia vyžadovala administrátorské práva, no bola spúšťaná iba s právami užívateľskými.

Druhým problematickým aspektom je problém s nastavením hostname. V prípade, že je ako hostname nastavený localhost, nastáva problém s loopbackom, ktorý primárne spôsobí zablokovanie

inštalácie. V prípade, že sa predsa len podarí túto chybu počas inštalácie potlačiť, zablokuje sa celý Grid engine pri naplánovaní prvého jobu. V takomto prípade daemon reaguje na každý príkaz chybovou hláškou. Keďže daemon reaguje v takomto prípade na každú operáciu chybou, nie je možné job zastaviť, zrušiť jeho naplánovanie, spustiť iný job, vypísať štatistiky a dokonca ani spustiť QMON. Platí pri tom, že vzhľadom na nedostatočnú dokumentáciu systému je snaha o opravenie chyby, v prípade, že nastane, častokrát veľmi obtiažnou úlohou. Nápomocné v tomto smere nie sú ani chybové hlášky, či výpisy v logu a jediným riešením sa tak stáva hlbšia znalosť systému.

Zmätočné verziovanie

Ďalší problematický aspekt predstavuje relatívny zmätok vo verziovaní niektorých vetiev Grid enginu. V tomto prípade predstavovalo veľký problém verziovanie SoGE. Keďže na vývoji a opravách chýb sa podieľa primárne Dave Love, ktorý spolupracuje s komunitou na opravách a riešení problémov s SoGE, v nepravidelných intervaloch vydáva aj opravy. Problém vo verziovaní spočíva v tom, že medzi ľuďmi, ktorí sa snažia o opravu SoGE sú aj takí, ktorí vlastnú upravenú verziu publikujú buď na stránkach rôznych git služieb (primárne GitLab a GitHub).

Týmto publikovaným verziám častokrát pridelia o 1 zvýšené číslo verzie. Takýmto spôsobom vzniká viacero rôznych riešení s rovnakým číslom verzie, pričom každá z nich obsahuje iné skryté chyby. Takéto verzie predstavujú problém hlavne v momente riešenia problémov, ktoré nastanú počas inštalácie, či behu enginu.

Problémy s podporou zo strany OS a Javy

Keďže testovaných bolo viacero verzii SoGE, okrem vyššie spomenutých chýb bol jedným z veľmi častých problémov podpora zo strany OS alebo Javy. U mnohých z testovaných verzii platilo, že z neznámych príčin boli schopné pracovať len na starších verziách OS, pričom na novších sa buď potýkali s problémami za behu alebo v horšom prípade nefungovali vôbec. Príčinou pritom bola častokrát verzia Javy. Najspoľahlivejšou bola z hľadiska verzii Javy verzia 1.8.0.

Z hľadiska verzii OS bolo odporúčanou verziou u väčšiny verzii SoGE práve centOS vo verzii 6 a predovšetkým 7, prípadne jeho Enterprise varianta Red Hat. Okrem toho sa našli aj verzie s podporou pre Debian, Ubuntu a OpenSuse, ktoré boli zároveň najobmedzenejšími z pohľadu požadovanej verzie systému.

4.2 Testovanie klienta

Ako bolo spomenuté v podkapitole 3.3, klient aplikácie bol vytváraný použitím JavaScriptového multiplatformového prostredia Node.js s využitím knižnice React. Medzi najznámejšie nástroje, resp. knižnice, ktoré pomáhajú analyzovať problémové časti kódu, debugovať kód efektívnejšie patrí nástroj ESLint, ktorý je dostupný výhradne vo forme knižnice pre Node.js aplikácie.[23]

Aj z toho dôvodu bol počas vývoja použitý práve ESLint a na časť testov bol použitý framework Selenium[24]. Najviac testov bolo však aj napriek tomu vykonaných ručným testovaním - zadávaním kritických a hraničných hodnôt, testovaním funkčnosti jednotlivých častí grafického rozhrania, ktoré prebiehalo prostredníctvom prehliadačov Google Chrome a Mozilla Firefox, a ich rozšírení.

Problémy pri vývoji knižnice

Medzi najproblematickejšie časti počas vývoja klienta patril vývoj knižnice umožňujúcej generovanie formulárov. Hlavné problémy spôsobovala pritom validácia vstupov pri opätovnom vyrendrovaní, či nahradení objektu JSON počas behu a rýchlosť generovania rozsiahlych formulárov. Na validáciu boli pôvodne použité iba metódy onChange a isValid. Keďže však nastávali problémy pri spätnom vygenerovaní formulára, kedy bez ohľadu na to, či bol objekt valídny alebo nie, vracala komponenta negatívny výsledok, bolo potrebné pridať validáciu aj do metódy getDerivedStateFromProps.

Rýchlosť načítania

Okrem testovania funkčnosti boli vykonané aj testy výkonu načítania jednotlivých komponentov GUI. Na testovanie boli použité vývojárske nástroje, ktoré poskytujú prehliadače, rovnako ako rozšírenie React Profiler a React Components slúžiace na analýzu aplikácii tvorených prostredníctvom knižnice React. Počas testovania sa ukázalo ako najdlhšie trvajúce práve prvé načítanie danej stránky, pri ktorom sa vykonal zápis do Cache prehliadača, čo málo za následok rýchlejšie načítanie v prípade ďalšieho načítania stránky, prípadne opakovaného načítania.

Obr. 4.1: Meranie rýchlosti načítavania GUI

Testovaním sa zistilo, že načítanie sa u jednotlivých stránok pohybovalo v rádoch sekúnd v prípade prvého načítania stránky. Doba načítania stránky sa v prípade druhej a ďalších odoziev pohybovala v rádoch niekoľkých milisekúnd, nakoľko pri prvom načítaní prebieha ukladanie súborov do cache prehliadača. Rýchlosť načítania stránky je pritom z veľkej časti závislá na odozve serveru

s API, ako aj na hardware, ktorým disponuje užívateľ, nakoľko rendrovanie GUI prebieha na strane užívateľa.

4.3 Výkonnostné testovanie

Jedná sa o typ testovania vykonávané za účelom určiť, ako sa systém správa pod určitou záťažou. Cieľom je stanoviť limity aplikácie, ale aj overiť, či aplikácia spĺňa požadované kritéria, kladené na výkon. Najväčší dôraz sa pritom kladie na parametre akými sú doba odozvy servera, na ktorom beží API a následný vplyv na HPC, na ktorom prebieha výpočet pipeline. Na testovanie bol v prípade výkonnostných testov použitý nástroj JMeter. Jedná sa o Java aplikáciu, používanú ako nástroj na testovanie záťaže, analýzu a meranie výkonu rôznych služieb u webových aplikácii.[25] Aby bol zistený dopad ako na výkon API, tak na výkon výpočtového zariadenia, ktoré s API komunikuje prostredníctvom SSH a SFTP, boli použité plug-iny SSHMon a PerfMon umožňujúce sledovanie výpočtového zariadenia cez SSH počas priebehu testov.

Popis testovacieho prostredia

Pre účely testovania boli predpripravené štyri typy transakcii, respektíve typov správania užívateľa:

- **Monitor host,Submit,Watch** - náhodne dlhú dobu sleduje základné informácie týkajúce sa hosta, vytvorí pipeline, zobrazí si detail, v ktorom pokračuje v sledovaní výkonu zariadenia
- **Extended monitor all** - užívateľ, ktorý monitoruje master hostov, execution hostov a fronty, pričom vrámci master hosta sleduje každé z jadier CPU, prípadne stav každej z partícii na disku
- **System administration** - simuluje správanie administrátora, ktorý vykonáva CRUD operácie nad pipeline template a užívateľmi
- **Submit,Just watching** - náhodne dlhú dobu sleduje základné informácie týkajúce sa hosta, vytvorí pipeline, zobrazí si detail, v ktorom sleduje aktuálny priebeh pipeline v jej detaile
- **Monitor** - sleduje aktuálne základné informácie týkajúce sa vyťaženie jednotlivých hostov

Keďže sa predpokladá, že mnohé zo skriptov budú bežať dlhšiu dobu, nakoľko budú pracovať s veľkým objemom dát a väčšinou bude systém prevažne slúžiť na sledovanie ich priebehu, najväčší počet užívateľov by malo vykonávať práve transakciu Submit,Just watching a Monitor. Naopak najmenší počet užívateľov by za normálnych okolností malo vykonávať transakciu Extended monitor all, ktorá pozlúži na otestovanie vplyvu detailného monitorovania HPC na jeho výkon.

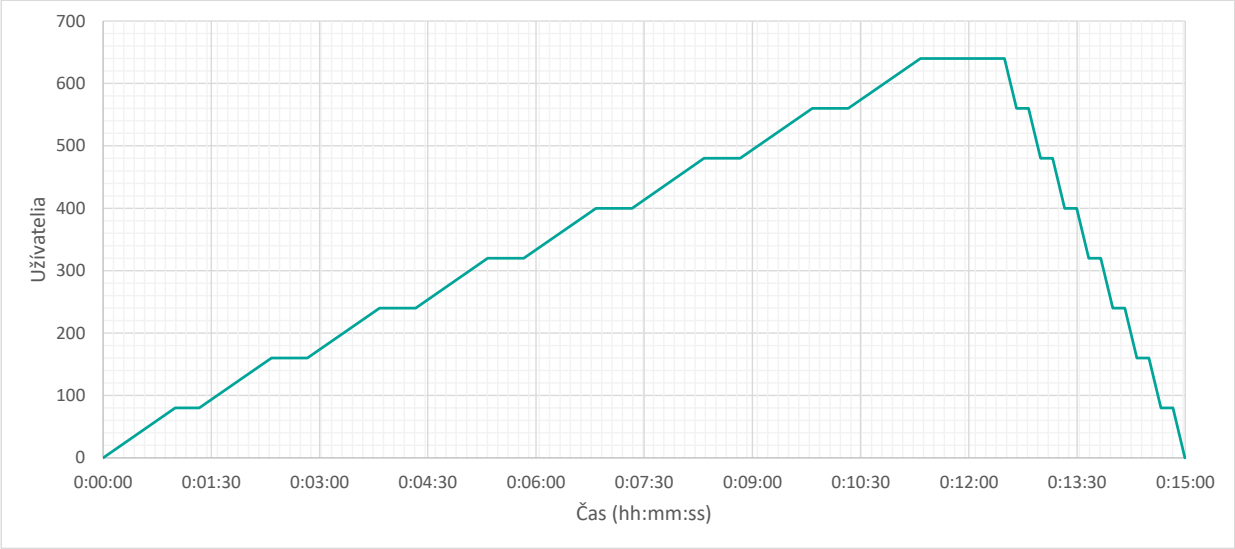
Názov	Počet HTTP Requestov
System administration	47
Monitor host,Submit,Watch	17
Monitor	8
Submit,Just watching	9
Extended monitor all	7

Tabuľka 4.2: Počet HTTP requestov vrámci jednotlivých transakcii

Počas testovania boli na zaznamenávanie dát týkajúcich sa výkonu a vyťaženia použité nasledujúce Listenery:

- **View Result Tree** - zobrazenie podrobných informácii týkajúcich sa jednotlivých HTTP Requestov
- **SSHMon Samples Collector** - graf využitia zdrojov servera (merané SSH pripojením)
- **Aggregate Report** - meranie priepustnosti, dĺžky odozvy, chybovosti a prenesených dát
- **Response Times Over Time** - graf dĺžky doby odozvy
- **PerfMon Metrics Collector** - graf využitia zdrojov servera
- **Active Threads Over Time** - graf počtu aktívnych pripojení

4.3.1 Stress test



Obr. 4.2: Priebeh zvyšovania záťaže počas stress testu - HPC VŠB

Test hraničnej záťaže označovaný aj ako Stress test je typ testovania, ktorý slúži na hľadanie maximálnych limitov aplikácie. Princíp spočíva v neustálom zvyšovaní záťaže, dokiaľ nedôjde k chybe, respektíve k pádu aplikácie v dôsledku preťaženia. V prípade webových aplikácii sa preťaženie obvykle prejavuje vysokou dobou odozvy v dôsledku neschopnosti spracovávať ďalšie požiadavky. Namerané hodnoty sú pritom vysoko závislé na zvolenom hardware.

Počas testovania bola použitá metóda Ramp up, aby sa zistili približné limity aplikácie. Nakoľko je aplikácia stavaná tak, aby bola schopná obsluhovať viaceré HPC, predpokladá sa, že by v priebehu času mohlo dôjsť k postupnému navyšovaniu užívateľov a tým aj požiadavok kladených na aplikáciu.

4.3.2 Vplyv monitorovania na zariadenie z Grid enginom

Jedným z cieľov stress testovania bolo zistiť, aký veľký vplyv má monitorovanie na zariadenie, na ktorom prebieha výpočet, prípadne na master hosta. Počas testovania bol každých 90 sekund navyšovaný počet užívateľov o 20 po dobu 15 minút, viď obrázok 4.2, čím sa dosiahlo maximálneho počtu 160 užívateľov v prípade HPC VŠB. V prípade VM s centOS sa počet zvyšoval vždy o 5 užívateľov, čím sa dosiahlo maximálneho počtu 40 užívateľov.

V prípade, že execution host nie je zároveň master host, monitorovanie nemá naň žiaden vplyv, nakoľko prenos dát týkajúcich sa vyťaženia medzi execution a master hostom prebieha prostredníctvom communication daemona.

Monitorovanie priamym pripojením k zariadeniu

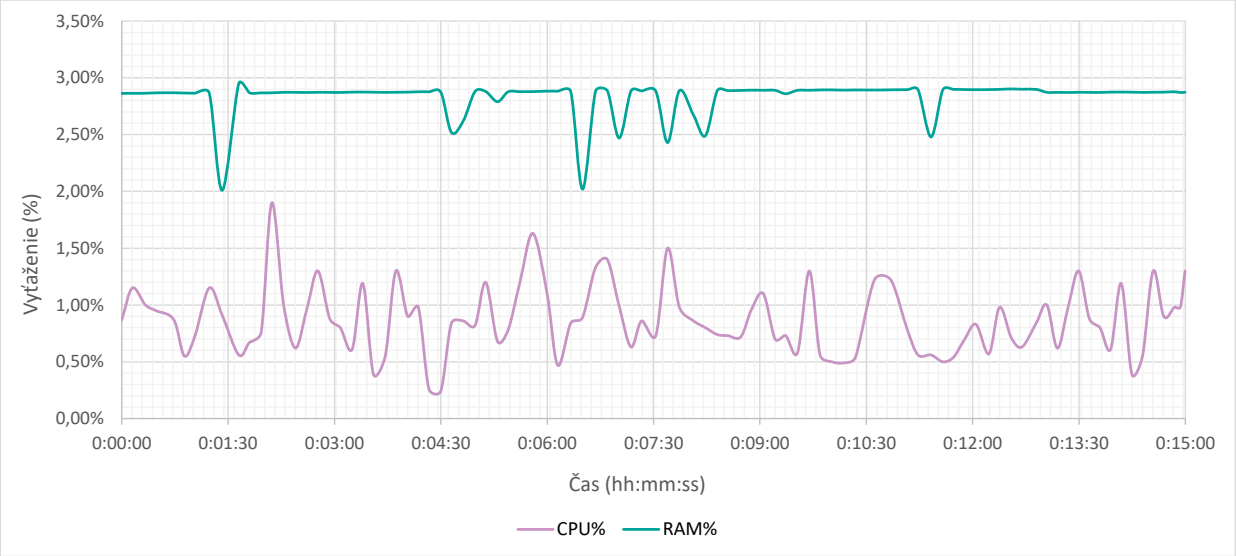
Počas testovania bola použitá transakcia Extended monitor all, ktorá získavá informácie týkajúce sa detailného vyťaženia priamo od zariadenia v momente, keď sú požadované. V oboch prípadoch prebiehalo pritom pripojenie prostredníctvom SSH.

Sledovanie vyťaženia v tomto prípade u zariadenia neprebieha na pozadí, nakoľko sa predpokladá, že tieto informácie bude chcieť získavať malé množstvo užívateľov - nanajvýš v rádoch jednotiek a veľmi ojedinele. Ich získavanie a spracovávanie na pozadí by zariadenie zbytočne vyťažovalo, nakoľko sa jedná o relatívne rozsiahle informácie, s nie až tak vysokým významom pre užívateľa.

- **HPC VŠB**

  Počas testovania sa ukázalo, že aj napriek vysokej záťaži bol server s API schopný spracovať a odpovedať na požiadavky užívateľa veľmi rýchlo aj napriek vysokému počtu užívateľov a komunikácii s HPC cez VPN. Pri vyššom počte užívateľov dochádzalo k veľmi rýchlemu spomaleniu odozvy v prípade požiadavok požadujúcich aktuálne informácie týkajúce sa vyťaženia jednotlivých CPU a spotreby jednotlivých partícii disku. Ostatné požiadavky zasielané na server však neboli ovplyvnené, viď obrázok 4.4.

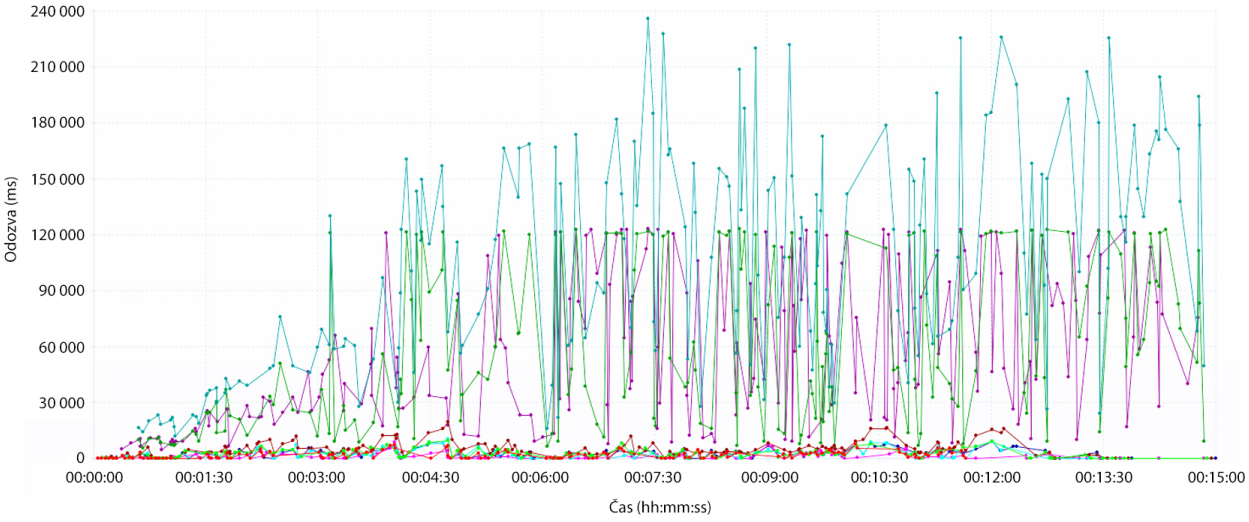
  Ako vidieť v tabuľke 4.3, počas testovania bolo poslaných celkovo 2 566 požiadavok, pričom u 7.366% došlo k chybe z dôvodu príliž dlhého čakania na odpoveď. Priemerná doba spracovania



Obr. 4.3: Stress test HPC VŠB počas detailného monitorovania - vyťaženie HPC VŠB

				Percentil (ms)			
Počet	**Priemer**	**Chyby**	**Priepustnosť**	**50%**	**90%**	**95%**	**99%**
2 566	25 093 ms	7.366%	2.94179 požiadavok/s	4 049	111 813	122 916	180 613

Tabuľka 4.3: Výsledky stress testu HPC VŠB počas monitorovania zariadenia



Obr. 4.4: Stress test HPC VŠB počas detailného monitorovania - vyťaženie API

požiadavky sa pritom vyšplhala až na približne 25 sekúnd. Ďalšie dôležité informácie poskytuje percentil, na základe ktorého môžno povedať, že 95% požiadavok bolo spracovaných serverom do 122 sekúnd, čo predstavuje odozvu asi 2 minúty. Ako však vidieť na obrázku 4.4, hodnoty odozvy dosahovali v najhorších prípadoch hodnôt až 240 000ms.

Pri pohľade na dáta o vyťažení získaných z výpočtového zariadenia je pritom vidieť, že vyťaženie CPU výpočtového zariadenia sa pohybovalo na úrovni 1% a RAM na úrovni 3%. Vyťaženie zariadenia sa veľmi nezvyšovalo ani s narastajúcim množstvom užívateľov, viď obrázok 4.3.

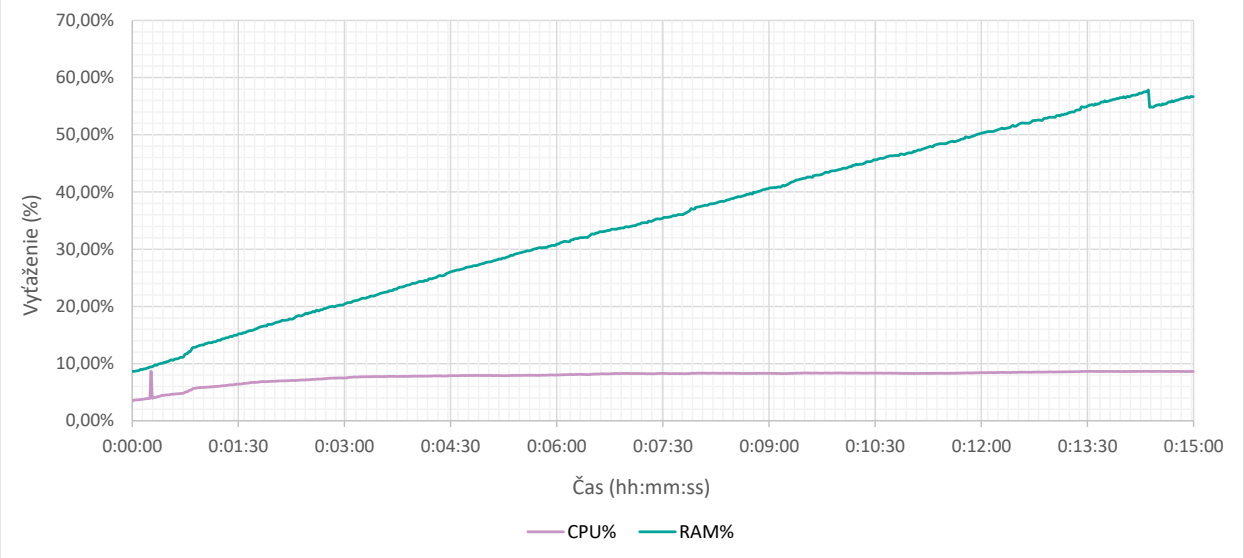
- **VM s centOS**

  Ako ukázal stress test u VM s centOS, v prípade zariadení s nižším, resp. s malým množstvom zdrojov dôjde veľmi rýchlo k maximálnemu vyťaženiu výpočtového zariadenia, viď obrázok 4.5. Dôvodom, takto rýchleho vyťaženia je z časti aj synchronizácia dát medzi Grid enginom a API prebiehajúca prostredníctvom službieb na pozadí.

  Komunikácia prebiehala v tomto prípade s master hostom Grid enginu, pričom na toto zariadenie bol napojený jeden execution host, ktorý nebol narozdiel od master hosta ovplyvnený.

  Počas testovania bolo poslaných celkovo 1 277 požiadavok, pričom u žiadnej z nich nedošlo k chybe. Priemerná doba spracovania požiadavky sa pritom vyšplhala rovnako ako v prípade testu s HPC až na približne 25 sekúnd. Priepustnosť v tomto prípade klesla na 1.4 požiadavky/s.

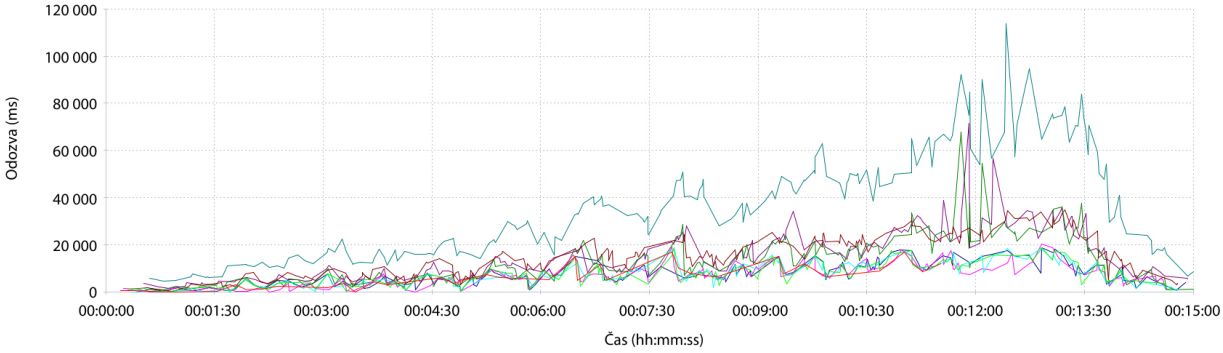
  Z obrázku 4.5 vyplýva, že najvyťaženejším zdrojom bola v tomto prípade RAM, ktorej spotreba sa blížila 60%.



Obr. 4.5: Stress test VM s centOS počas detailného monitorovania - vyťaženie VM s centOS

				Percentil (ms)			
Počet	**Priemer**	**Chyby**	**Priepustnosť**	**50%**	**90%**	**95%**	**99%**
1 277	24 577 ms	0%	1.4 požiadavok/s	6 814	84 255	95 640	168 563

Tabuľka 4.4: Výsledky stress testu VM s centOS počas detailného monitorovania



Obr. 4.6: Stress test VM s centOS počas detailného monitorovania - vyťaženie API

Monitorovanie zariadenia na pozadí

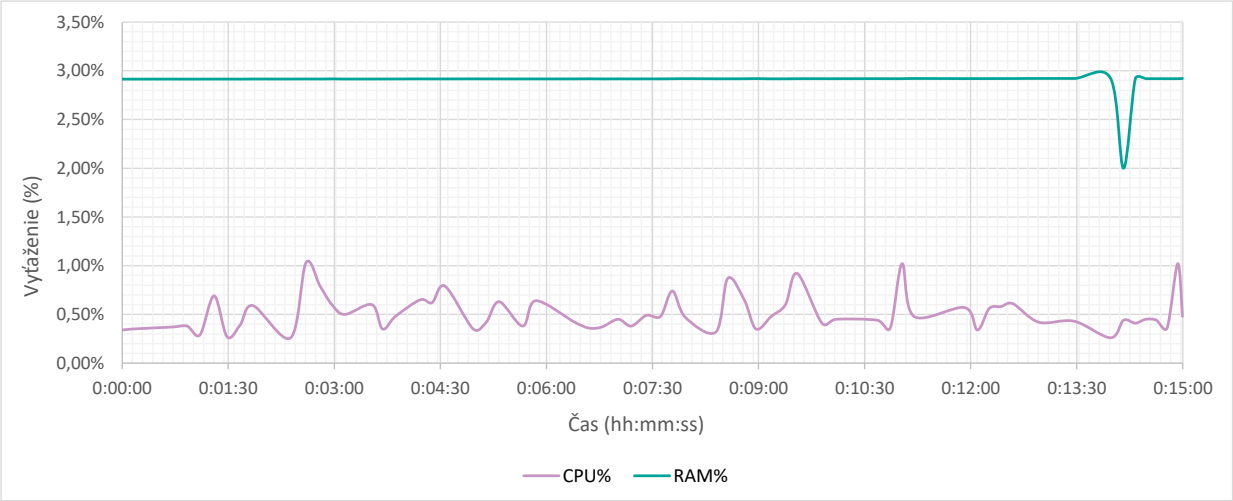
Okrem testovania monitorovania priamym spojením bolo testované aj vyťaženie spôsobené synchronizáciou bežiacou na pozadí. Počas testovania boli použité všetky transakcie, okrem transakcie Extended monitor all, ktorá vykonáva monitorovanie zariadenia prostredníctvom priameho pripojenia, ktoré neprebieha na pozadí API.

U tohto typu monitorovania sa na rozdiel od už testovaného predpokladá, že tieto informácie bude chcieť získavať veľké množstvo užívateľov hlavne počas monitorovania pred a po naplánovaní pipeline. Prebieha na pozadí prostredníctvom služby, ktorá získavá dáta týkajúce sa vyťaženia od Grid enginu.

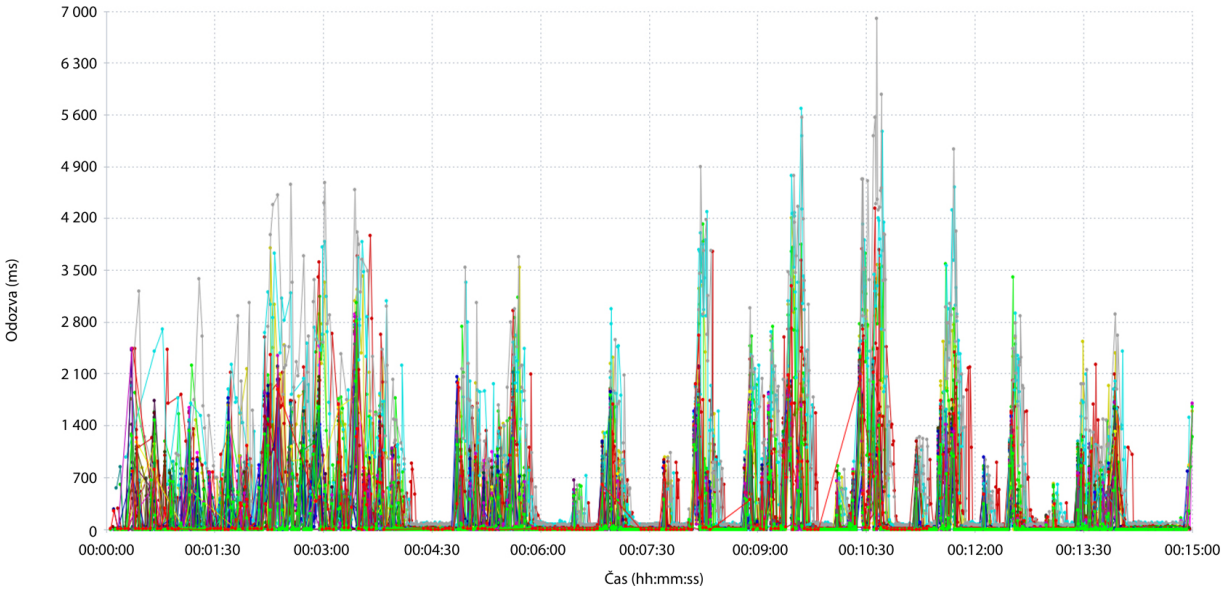
- **HPC VŠB**

  Počas testovania sa ukázalo, že bol server s API schopný spracovať a odpovedať na požiadavky užívateľa veľmi rýchlo aj napriek vysokému počtu užívateľov a komunikácii s HPC cez VPN. Pri vyššom počte užívateľov dochádzalo k relatívne pomalému spomaleniu odozvy.

  Ako vidieť v tabuľke 4.5, hodnoty odozvy dosahovali v priemerne 174 ms, pričom počas testu bolo poslaných celkovo 106 528 žiadostí. Na základe percentilu možno povedať, že až 99% žiadostí bolo spracovaných do 2 159ms. V grafe na obrázku 4.8 možno vidieť časti, v ktorých došlo k zvýšeniu doby odozvy. Vo väčšine prípadov z dôvodu zasielania alebo prijímania veľkých objektov - konkrétne sa tak deje obvykle v prípade vytvárania, prípadne editácie skriptov a pipeline templatov.



Obr. 4.7: Stress test HPC VŠB bez detailného monitorovania - vyťaženie HPC VŠB



Obr. 4.8: Stress test HPC VŠB bez detailného monitorovania - vyťaženie API

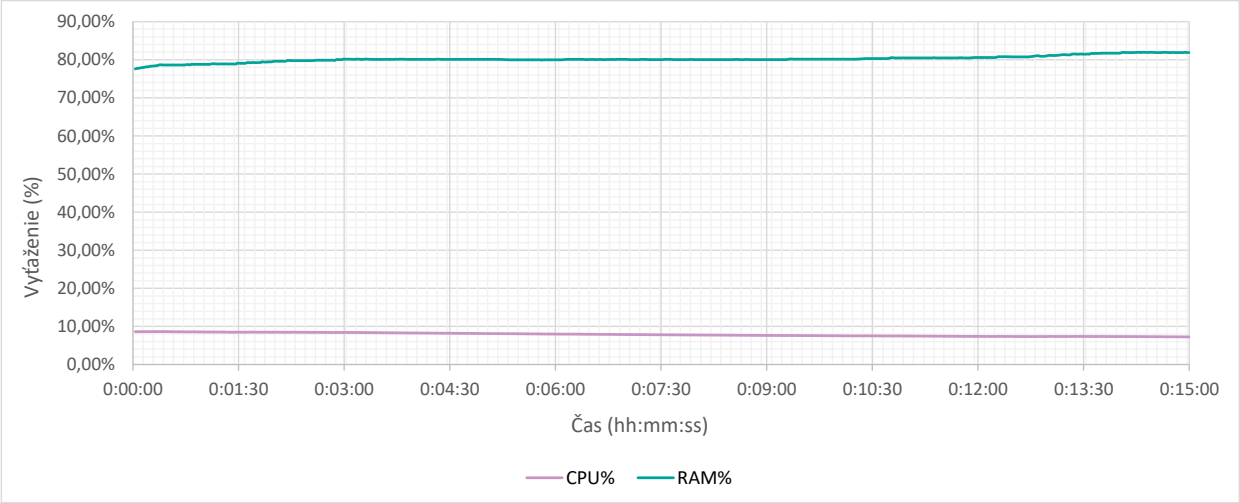
				Percentil (ms)			
Počet	**Priemer**	**Chyby**	**Priepustnosť**	**50%**	**90%**	**95%**	**99%**
106 528	174 ms	0%	118.8 požiadavok/s	12	626	1135	2159

Tabuľka 4.5: Výsledky stress HPC VŠB testovania bez detailného monitorovania

Pri pohľade na dáta o vyťažení získaných z výpočtového zariadenia je vidieť, že vyťaženie CPU sa pohybovalo približne na rovnakej úrovni ako v prípade predchádzajúceho testu. Vyťaženie zariadenia sa veľmi nezvyšovalo ani s narastajúcim množstvom užívateľov, viď obrázok 4.7.

- **VM s centOS**

  Na rozdiel od predchádzajúcich testov test na VM s centOS ukázal, že v prípade zariadení s nižším, resp. s malým množstvom zdrojov, dôjde veľmi rýchlo k maximálnemu vyťaženiu výpočtového zariadenia, viď obrázok 4.9. Pričom dôvodom takto rýchleho nárastu vyťaženia je z časti aj synchronizácia dát medzi Grid enginom a API prebiehajúca prostredníctvom služby na pozadí.



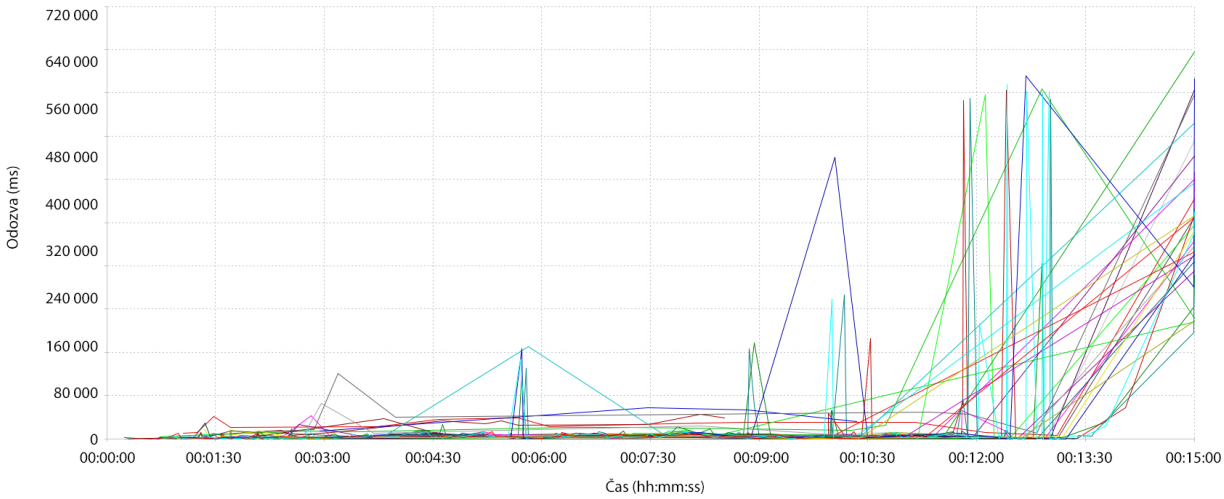
Obr. 4.9: Stress test VM s centOS bez detailného monitorovania - vyťaženie VM s centOS

Komunikácia prebiehala v tomto prípade s master hostom Grid enginu, pričom na toto zariadenie bol napojený jeden execution host, ktorý nebol na rozdiel od master hosta ovplyvnený.

Z tabuľky 4.6 tiež možno usúdiť, že počas testu dosiahla chybovosť hodnotu 8.204%, pričom hlavným dôvodom bol nedostatok zdrojov na strane výpočtového servera. Na základe obrázka 4.10 však možno povedať, že nedostatok zdrojov sa objavil až v závere testovania, keď bol počet pripojených užívateľov najväčší. Priemerná odozva v tomto prípade dosahovala síce 45 479ms, avšak pri pohľade na percentil možno povedať, že u 95% požiadavok trvala odpoveď servera až 399 999ms.

				Percentil (ms)			
Počet	**Priemer**	**Chyby**	**Priepustnosť**	**50%**	**90%**	**95%**	**99%**
1926	45 479 ms	8.204%	2.14334 požiadavok/s	4 880	53 754	399 999	653 778

Tabuľka 4.6: Výsledky stress testu VM s centOS bez detailného monitorovania



Obr. 4.10: Stress test VM s centOS bez detailného monitorovania - vyťaženie API

4.3.3 Load test

Na rozdiel od stress testov sa v prípade load testovania jedná o sledovanie na určitej (predpokladanej) úrovni záťaže s cieľom simulovať reálne podmienky kladené na aplikáciu. Použitou metódou bude v tomto prípade Big Bang, u ktorej na rozdiel od metód použitých pri stress testoch je počet užívateľov počas celej doby testu nemenný a všetci užívatelia sú spúšťaní naraz na začiatku.

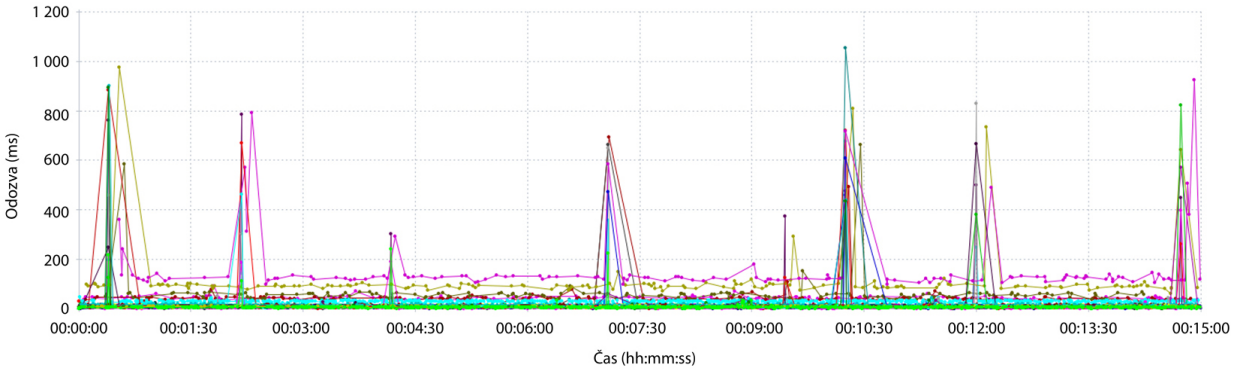
Transakcia	**Počet užívateľov**
Monitor host,Submit,Watch	5
Extended monitor all	2
System administration	4
Submit,Just watching	5
Monitor	20
Spolu	37

Tabuľka 4.7: Predpokladané hodnoty záťaže - load test

- **HPC VŠB**

  Ako vidieť v tabuľke 4.8, počas testovania bolo poslaných celkovo 11 078 požiadavok, pričom u žiadnej z nich nedošlo k chybe. Priemerná doba potrebná na spracovanie požiadavky bola len 20 ms. Ďalšie dôležité informácie poskytuje percentil, na základe ktorého možno povedať, že 95% požiadavok bolo spracovaných serverom do 48 ms.

  Na obrázku 4.11 vidieť niekoľko bodov, v ktorých došlo k relatívne výraznému predĺženiu doby odozvy. K tomuto správaniu dochádzalo v dobe vytvárania a úpravy pipeline templatov



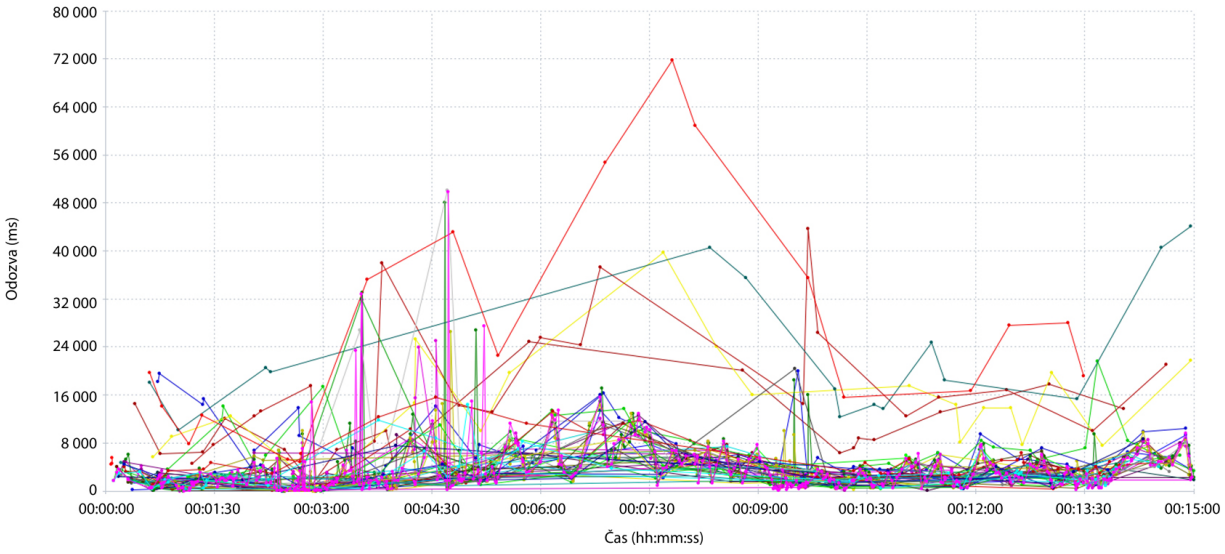
Obr. 4.11: Stress test - vyťaženie API

a skriptov, pretože počas týchto operácii prebiehajú najväčšie prenosy medzi klientom a API. Zároveň pri vytváraní dochádza k relatívne veľkým prenosom dát medzi databázou a API, a v prípade vytvárania skriptu aj k zápisu na disk.

				Percentil (ms)			
Počet	**Priemer**	**Chyby**	**Priepustnosť**	**50%**	**90%**	**95%**	**99%**
11078	20 ms	0%	12.31012 požiadavok/s	11	34	48	133

Tabuľka 4.8: Výsledky stress testovania monitorovania zariadenia - HPC VŠB

- **VM s centOS**



Obr. 4.12: Stress test - vyťaženie API

				Percentil (ms)			
Počet	**Priemer**	**Chyby**	**Priepustnosť**	**50%**	**90%**	**95%**	**99%**
4 128	3 893 ms	0%	4.5768 požiadavok/s	2 696	7 595	10 387	22 652

Tabuľka 4.9: Výsledky stress testovania monitorovania zariadenia - VM s centOS

Ako vidieť v tabuľke 4.9, počas testovania bolo poslaných celkovo 4 128 požiadavok, pričom aj v tomto prípade dosahovala síce chybovosť hodnotu 0%, avšak dosahoval priepustnosť len 4.5768 požiadavok/s, čo je približne dvojnásobok oproti výsledku dosiahnutému počas stress testu.

Na obrázku 4.12 vidieť mnoho bodov, v ktorých došlo k veľmi výraznému predĺženiu doby odozvy. K tomuto správaniu dochádzalo v dobe vytvárania a úpravy pipeline templatov a skriptov, z dôvodu nedostatku zdrojov na strane servera s Grid enginom.

4.3.4 Rýchlosť naplánovania pipeline

Počas stress a load testovania bolo vykonané aj testovanie, ktoré by malo ukázať, ako sa bude meniť doba potrebná na prípravu a naplánovanie pipeline v prípade nárastu požiadaviek zo strany užívateľov. Táto doba je závislá na nastavení DataTransferService a hardware zariadenia, na ktorom beží API. Keďže plánovanie a presun dát na zariadenie prebieha na pozadí, dĺžka prípravy dát na prenos nevykazovala žiadne extrémne odchýlky s rastúcim množstvom požiadavok na naplánovanie novej pipeline.

Zmeny boli badateľné však počas stress testovania u doby potrebnej na naplánovanie na strane Grid enginu, ktorá s rastúcim množstvom požiadavok úmerne rástla a v najhoršom prípade dosahovala hranicu takmer 3 minút. Doba je závislá na nastavení časového intervalu a počtu taskov spracovávaných paralelne vrámci služby DataTransferService, viď tabuľka 4.10.

			Percentil (ms)			
Test	**Počet**	**Priemer**	**50%**	**90%**	**95%**	**99%**
Stress	121	74 054 ms	67 493	134 348	138 027	140 872
Load	13	15 540 ms	15 500	22 600	23 450	23 450

Tabuľka 4.10: Rýchlosť naplánovania pipeline počas stress testovania

Jedná sa o očakávané správanie zapríčinené implementáciou. Realizované riešenie je totiž naprogramované tak, aby chránilo zariadenie, na ktorom prebieha plánovanie, prípadne aj výpočet na úkor rýchlosti naplánovania. V prípade HPC VŠB malo svoj podiel na rýchlosti plánovania aj pripojenie prebiehajúce cez VNP. V prípade load testovania sa doba medzi naplánovaním na strane klienta a naplánovaním na strane Grid engine pohybovala v rozmedzí 15 až 30 sekúnd.

Kapitola 5

Záver

Ako bolo spomenuté už v úvodnej časti práce, cieľom bolo vytvorenie webovej aplikácie pre podporu analýzy dát, špeciálne biomedicínskych dát, ktorá je zároveň hlavným výsledkom tejto diplomovej práce.

Keďže bolo jednou z hlavných požiadavok, aby aplikácia umožňovala jednoduché a efektívne plánovanie jobov v prostredí HPC, bol vykonaný prieskum plánovačov rodiny Grid engine. Špeciálna pozornosť bola venovaná preskúmaniu celkového fungovania SoGE, ktorý predstavuje spomedzi plánovačov jobov jednu z najlepších volieb, nakoľko sa jedná o jedno z mála open source riešení patriacich do tejto rodiny, na ktorom sa stále pracuje. Zistilo sa, že všetky plánovače z tejto rodiny využívajú tie isté príkazy, čím sa zachováva akási jednota a spätná kompatibilita, čo zabezpečuje kompatibilitu vytvoreného riešenia prakticky s ktorýmkoľvek z potomkov Grid engine - teda aj s UGE, OGE a ďalšími. Zároveň sa zistilo, že väčšina dokumentácie je buď zastaralá alebo nedostupná. Značná časť verzii obsahuje relatívne veľké množstvo chýb a zraniteľností, ku ktorým dokumentácia ani príspevky na fórach neexistujú. Ako sa hovorí aj v kapitole 4.1, už samotná inštalácia predstavovala relatívne obtiažnu úlohu, nakoľko ju komplikoval zmätok vo verziách, ale aj závislosť každej z verzii na konkrétnej verzii OS a verzii Java.

Pretože každý skript má iný počet a typy vstupných parametrov, bolo potrebné vrámci vývoja aplikácie vyriešiť spôsob, akým generovať formuláre na zadávanie vstupov. Bol vykonaný prieskum knižníc, ktoré umožňujú generovať formuláre z objektov JSON, ktorý bol zvolený hlavne kvôli jednoduchšej práci vrámci jazyka JavaScript, v ktorom bola tvorená klientská časť aplikácie. Ten ukázal, že počet knižníc poskytujúcich takúto funkcionalitu nie je veľmi veľký. Väčšina z nich pritom, ako uvádza kapitola 2.3, poskytuje rovnakú mieru funkčnosti. Žiadna z nich však neposkytuje popri generátore formulárov dizajnér, ktorý by mohol jej užívateľ použiť vo svojej aplikácii. Preto v tomto smere vyniká nami vytváraná knižnica, ktorej vývoj, celkové výhody a nevýhody sú popísané v kapitole 3.5. Poskytuje totiž široké možnosti nastaveni, medzi ktoré patrí napríklad voľba štruktúry vracaného JSON objektu, už spomínaný dizajnér, či možnosť preložiť si dizajnér do vlastného jazyka.

Po vyriešení problémov s konfiguráciou a inštaláciou SoGE ako aj vytvorení knižnice umožňu-

júcej generovať formuláre, bolo potrebné nájsť spôsob, akým efektívne plánovať pipeline tvorenú jobmi, ktoré môžu na seba rôznym spôsobom nadväzovať, pričom niektoré z nich môžu bežať paralelne a iné len sériovo za sebou. Pričom každý zo spúšťaných skriptov môže mať iné požiadavky na hardware alebo software, či už spomínané vstupné parametre, ktoré zadá užívateľ prostredníctvom vygenerovaného formulára. Údaje zaslané klientom vrámci servera s API spracovať, zostaviť potrebný bash skript, ktorý sa spolu so skriptami, eventuálne aj dátami zašle na vykonanie výpočtovému serveru tak, ako je popísané v kapitole 3.4.2. Po naplánovaní následne priebeh pipeline sledovať a po dokončení výstupné dáta sprístupniť užívateľovi, ktorý pipeline naplánoval. Na základe prieskumu z kapitoly 2.2 sme sa rozhodli pre použitie komunikácie prostredníctvom príkazov zasielaných do prostredia terminálu, vďaka čomu sa nám otvorila možnosť komunikovať so zariadením, na ktorom beží Grid engine tiež vzdialene prostredníctvom SSH.

Ďalší problém, ktorý bolo potrebné vyriešiť, bola otázka bezpečnosti spúšťaných pipeline. Užívateľ aplikácie totiž môže aplikáciou poslať na vykonanie škodlivý skript. Z dôvodu veľkej obtiažnosti detekcie škodlivých skriptov bolo rozhodnuté, že najlepším riešením bude dôvera užívateľovi a využitie systému prideľovania front skupinám užívateľov vrámci Grid enginu v závislosti na dôveryhodnosti. Správca teda môže zadefinovať maximálne množstvo zdrojov, s ktorými bude môcť užívateľov skript pracovať. V súvislosti s bezpečnosťou bolo tiež potrebné zabezpečiť, aby sa rôzne pipeline navzájom neovplyvňovali. V tomto prípade bola za riešenie zvolená utilita setAcl, ktorá minimalizuje dopad jednej pipeline na druhú v prípade, že pipeline patria rôznym užívateľom.

V neposlednom rade bolo potrebné nájsť spôsob ako efektívne sledovať dostupné zdroje a vyťaženosť zariadenia. Bolo pritom rozhodnuté, že najdôležitejšie informácie týkajúce sa zariadenia budú získavané na pozadí z prostredia Grid enginu, ktorý poskytuje aj informácie týkajúce sa execution hostov a to bez toho, aby bol akokoľvek ovplyvnený výkon zariadenia, na ktorom prebieha výpočet.

Výsledkom práce je plne funkčná webová aplikácia poskytujúca užívateľovi možnosť vytvárať, spúšťať a sledovať pipeline tvorené skriptami, ktoré sú plánované a spúšťané prostredníctvom Grid enginu. Monitorovať dostupné a spotrebované zdroje HPC a aktuálne dianie vo frontách a na hostoch s možnosťou napojenia na viaceré navzájom nezávislé HPC s Grid enginom. Do budúcna by som mal záujem ďalej rozširovať nielen vytvorenú knižnicu, ale aj vytvorenú webovú aplikáciu. Nakoľko u každej aplikácie existuje priestor ako na zlepšenie, tak na rozšírenie.

Literatura

1. DAGDIGIAN, Chris. *Which Grid Engine?* [online]. Bio-IT World [cit. 2020-01-20]. Dostupné z: http://www.bio-itworld.com/2012/02/15/which-grid-engine.html.
2. *What is Azure CycleCloud?* [online]. Microsoft [cit. 2020-01-20]. Dostupné z: https://docs.microsoft.com/en-us/azure/cyclecloud/overview.
3. *Ready-to-run Grid Engine clusters in AWS Marketplace* [online]. Amazon [cit. 2020-01-20]. Dostupné z: https://blogs.univa.com/2018/02/ready-to-run-grid-engine-clusters-in-aws-marketplace/.
4. *ElastiCluster* [online] [cit. 2020-01-20]. Dostupné z: https://elasticluster.readthedocs.io/en/latest/.
5. *History of Grid Engine Development* [online] [cit. 2019-11-05]. Dostupné z: http://www.softpanorama.org/HPC/Grid_engine/history.shtml.
6. MORGAN, Timothy Prickett. *Univa forks Oracle's Sun Grid Engine* [online] [cit. 2019-11-06]. Dostupné z: https://www.theregister.co.uk/2011/01/18/univa_forks_oracle_grid_engine/.
7. *Son of Grid Engine* [online] [cit. 2019-11-05]. Dostupné z: https://arc.liv.ac.uk/trac/SGE.
8. *Sun^{TM} ONE Grid Engine - Administration and User's Guide* [online]. Sun Microsystems, Inc., 2002 [cit. 2020-01-15]. Dostupné z: http://research.cs.rutgers.edu/~sgeadmin/SGE53AdminUserDoc.pdf.
9. DANIEL TEMPLETON, Sun Microsystems Inc. *ADVANCED SUN GRID ENGINE ADMINISTRATION* [online] [cit. 2020-02-02]. Dostupné z: https://wiki.chipp.ch/twiki/pub/CmsTier3/SunGridEngine/SGE-GeorgetownClass3.pdf.
10. ENGINEERING, Univa. *Grid Engine Users's Guide* [online]. Verzia 8.5.4 [cit. 2020-01-14]. Dostupné z: http://www.univa.com/resources/files/univa_user_guide_univa__grid_engine_854.pdf.
11. *Beginner's Guide to Oracle Grid Engine 6.2* [online]. Oracle Corporation, 2010 [cit. 2020-01-18]. Dostupné z: https://www.oracle.com/technetwork/oem/host-server-mgmt/twp-gridengine-beginner-167116.pdf.

12. CANNON, Brett. *Porting Python 2 Code to Python 3* [online] [cit. 2020-01-15]. Dostupné z: https://docs.python.org/3/howto/pyporting.html.

13. *2to3 - Automated Python 2 to 3 code translation* [online] [cit. 2020-01-15]. Dostupné z: https://docs.python.org/2/library/2to3.html.

14. *Distributed Resource Management Application API Version 2 (DRMAA)* [online]. 2012 [cit. 2019-11-01]. Dostupné z: https://www.ogf.org/documents/GFD.194.pdf.

15. *GitHub - RicherMans/QPy: A Gridengine wrapper for Python which easily allows to run functions on the Grid* [online] [cit. 2019-10-29]. Dostupné z: https://github.com/RicherMans/QPy.

16. BLAISE BARNEY, Lawrence Livermore National Laboratory. *Message Passing Interface (MPI)* [online] [cit. 2019-11-02]. Dostupné z: https://computing.llnl.gov/tutorials/mpi/.

17. *RUNNING JOBS ON THE HPC CLUSTER* [online] [cit. 2020-03-12]. Dostupné z: https://hpc.oit.uci.edu/running-jobs#_mpi.

18. *JSON Forms* [online] [cit. 2019-12-11]. Dostupné z: https://jsonforms.io/docs/.

19. *@jsonforms/react - npm* [online] [cit. 2019-12-11]. Dostupné z: https://www.npmjs.com/package/@jsonforms/react.

20. *react-jsonschema-form documentation* [online] [cit. 2019-12-13]. Dostupné z: https://react-jsonschema-form.readthedocs.io/en/latest/.

21. *Winterfell* [online] [cit. 2020-01-28]. Dostupné z: https://www.npmjs.com/package/winterfell.

22. *React JS Notes for Professionals book* [online] [cit. 2020-02-07]. Dostupné z: https://books.goalkicker.com/ReactJSBook/.

23. *ESLint / Pluggble JavaScript linter* [online] [cit. 2020-04-22]. Dostupné z: https://eslint.org/.

24. *SeleniumHQ Browser Automation* [online] [cit. 2020-04-27]. Dostupné z: https://www.selenium.dev/.

25. *Apache JMeter* [online] [cit. 2020-11-02]. Dostupné z: https://jmeter.apache.org/.

Dodatok A

Register pojmov

job	–	program naplánovaný a bežiaci na pozadí, spravidla bez zásahu užívateľa
skript	–	program alebo jeho fragment zapísaný v podobe kódu do súboru
pipeline	–	postupnosť jobov, ktoré na seba istým spôsobom naväzujú, pričom môžu bežať paralelne alebo sériovo v závislosti na požiadavkách užívateľa
pipeline template	–	predpripravená postupnosť skriptov, ktorá je ukladaná do databázy a môže byť naplánovaná v budúcnosti